University of Alberta

Mizar Verification of Algorithms for Recognizing Chordal Graphs

by

Broderick Arneson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-29932-6
Our file Notre référence
ISBN: 978-0-494-29932-6

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licencè non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

The objective of this work is to test the suitability of the Mizar proof assistant for automated proof checking of some select graph algorithms as they are published in research papers and survey books.

We use Mizar to check the proof of a characterization of chordal graphs, as well as the proofs of correctness of two algorithms for recognizing chordal graphs: Lexicographic Breadth First Search and Maximum Cardinality Search. This required the addition of some foundational results to the Mizar Mathematical Library. In the process of formalizing the graph algorithms we discovered some small inaccuracies in the published proofs.

Contents

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Formalized Mathematics . . . 1
- 1.3 MIZAR . . . 2
- 1.4 Related Works . . . 3
- 1.5 Organization of the Paper . . . 3

2 The MIZAR Graph Library 5
- 2.1 Graphs . . . 5
- 2.2 Labeled Graphs . . . 7
- 2.3 Subgraphs . . . 8
- 2.4 Connectivity . . . 10
- 2.5 Walks and Paths . . . 10
- 2.6 Graph Sequences/Algorithms . . . 11

3 Formalized Proofs 12
- 3.1 Characterization of Chordal Graphs . . . 12
  - 3.1.1 Definitions . . . 12
  - 3.1.2 Characterization . . . 15
  - 3.1.3 Discussion . . . 18
- 3.2 Vertex Numbering Graph Sequences . . . 18
- 3.3 Lexicographical Breadth-first Search Algorithm . . . 22
  - 3.3.1 Recognizing Chordal Graphs . . . 22
  - 3.3.2 The LexBFS algorithm . . . 22
  - 3.3.3 Formal Definitions of LexBFS . . . 23
  - 3.3.4 Proving Correctness of LexBFS . . . 27
  - 3.3.5 Discussion . . . 32
- 3.4 Maximum Cardinality Search . . . 32
  - 3.4.1 The MCS algorithm . . . 32
  - 3.4.2 Formal Definitions of MCS . . . 33
  - 3.4.3 Proving Correctness of MCS . . . 36
- 3.5 Discussion . . . 40

4 Conclusion and Future Work 41

Bibliography 42

A Introduction 44

B Triangulated Graphs: Golumbic **45**
1. Introduction 45
2. Characterizing Triangulated Graphs 45
3. Recognizing Triangulated Graphs by Lexicographic Breadth-First Search 49

C Triangulated Graphs: Revised **46**
1. Introduction 46
2. Characterizing Triangulated Graphs 46
3. Recognizing Triangulated Graphs by Lexicographic Breadth-First Search 50
4. Addendum 57

D MIZAR abstracts **60**
1. chord.abs 61
2. lexbfs.abs 68

Chapter 1

Introduction

1.1 Overview

This thesis tests the suitability of the MIZAR proof assistant for automated proof checking of some graph algorithms as they are published in research papers and survey books. We have chosen the development of chordal graphs as presented in the well known book by Golumbic[10].

Our goal consists in MIZAR proof-checking for correctness two algorithms for recognizing chordal graphs: Lexicographic Breadth First Search and Maximum Cardinality Search. The starting point of our work was the MIZAR Graph Library prepared by Gilbert Lee. His work in turn was based on the Mizar Mathematical Library developed over the last twenty years by dozens of people.

We consider our effort a success — we needed to add some foundational developments to the Graph Library and some minor facts to the Mathematical Library, but encountered no insurmountable obstacles during our work. In the process of formalizing the graph algorithms, we discovered some small inaccuracies in the published proofs. While our ambition was to follow the published proofs as closely as possible, we have departed from them at some places for the sake of simplicity. Indeed, informal proofs sometimes tend to gloss over technical difficulties when closer inspection reveals a simpler solution. The cost of MIZAR formalization, while substantial, is not overwhelming: formal proofs usually contain around ten times the number of tokens than their published counterparts.

1.2 Formalized Mathematics

One of the goals of formalized mathematics is to take mathematical knowledge and express it in such a way that automated processing is possible. Not only are such formalized proofs completely rigorous and verifiable by machine, but taken together they form a searchable database. There is hope that such a database could be mined for new, previously unknown results, or prove useful to automated theorem provers. Currently, there are quite a number of automated proof assistants in use around the world, such as ACL2[1], COQ[2], HOL[3], MIZAR[4], and PVS[5].

1.3 MIZAR

MIZAR, the proof checker system used in this project, is one of the oldest such systems still actively used. It was started in 1973 by Andrzej Trybulec. MIZAR consists of two main components, the verifier and the MIZAR Mathematical Libarary (MML). The verifier takes a file written in MIZAR syntax and checks it for logical errors. The MML is a database of MIZAR articles. The MML is built on the axioms of Tarski-Grothendieck set theory, and MIZAR's proof checker is based on classical first-order logic, furnished with some machinery for forming infinite schemes of statements.

We now present a simple proof, followed by a step by step explanation, to introduce the reader to MIZAR syntax. The proof shows that a complete graph is also a chordal graph. A graph is complete if it contains an edge between every pair of distinct vertices. A graph is chordal if every cycle with length greater than three has a chord, that is, every cycle contains an edge between non consecutive vertices of the cycle (see Definition 3.1.10 for the precise definition of a chord).

```
theorem
  for G being complete _Graph holds G is chordal
proof
  let G be complete _Graph;

    for W being Walk of G such that W.length() > 3 & W is Cycle-like
      holds W is chordal
    proof
        let W be Walk of G such that
    A2: W.length() > 3 & W is Cycle-like;

    A:  W is Path-like by A2,GLIB_001:def 31;
        W.length() >= 3+1 by A2,NAT_1:38; then
        2*W.length() >= 2*4 by XREAL_1:66; then
        2*W.length() + 1 >= 8 + 1 by XREAL_1:9; then
    A3: len W >= 9 by GLIB_001:113;

        reconsider t3=2*1+1 as odd natural number;
        reconsider t7=2*3+1 as odd natural number;
        t3 <= len W by A3,XREAL_1:2; then
        reconsider W3=W.t3 as Vertex of G by GLIB_001:8;
    A5: t7 <= len W by A3,XREAL_1:2; then
        reconsider W7=W.t7 as Vertex of G by GLIB_001:8;

        W3 <> W7 by A, A5,GLIB_001:def 28;
        then W3,W7 are_adjacent by DefComplete; then
        consider e being set such that
    A4: e Joins W3,W7,G by DefAdjacent;

        t3+2 < t7 & t7 <= len W & not (t3=3 & t7 = len W) by A3,XREAL_1:2;
      hence W is chordal by A, A4,ChordalPath01;
    end;

  hence G is chordal by DefChordalGraph;
end;
```

The statement we wish to prove follows the `theorem` keyword and says that for any complete graph `G` it holds that `G` is chordal. The proof is contained in the code block beginning after `proof` and ending before the last `end`. In the first line of the

proof we consider an arbitrary complete graph G. In order to prove that G is chordal, we need to show that every cycle with length greater than three possesses a chord, in other words, that every such cycle is chordal. Notice that we can use attributes of the same name on objects of different types with no difficulty (here we have chordal graphs and chordal walks). To prove this new statement we open a new proof block. In the first line of this new proof we consider an arbitrary cycle W of G with length greater than three. W is defined as an alternating sequence of vertices and edges, and so the elements with odd indices in W are vertices and the elements with even indices are the edges joining these vertices. The `length()` function returns the number of edges in W, and the `len` W function returns the number of vertices plus the number edges. For a more precise discussion of how walks are implemented please see Section 2.5. We show the two elements at indices 3 and 7 are vertices of G and refer to them as W3 and W7 respectively. W3 does not equal W7 because a cycle is assumed to repeat only the first and last vertices; since W3 and W7 are not the the first and last vertices they are not equal. Since all distinct vertices are adjacent in a complete graph, we can consider the edge e joining W3 and W7. But e is a chord of W, and so W is chordal. Hence G is a chordal graph.

1.4 Related Works

In 1990, Hryniewiecki[11] formalized some basic graph structures in MIZAR, which was followed subsequently by articles from Rudnicki, Nakamura, and Chen[20, 21, 22, 23]. Chen also showed a proof of correctness for Dijkstra's algorithm [8], using an approach completely different from the type of approach we will use for graph algorithms. Chen simulates the effects of the algorithm on an array holding the graph information. Our approach is to use basic graph operations to create a sequence of graphs representing the steps of the algorithm through time. In 2004, Gilbert Lee started the graph library that we will extend in this thesis, and proved the correctness of Dijkstra's Single Source Shortest Path, Prim's Minimum Spanning Tree and Ford/Fulkerson Max-Flow algorithms [12, 17, 18, 15, 16, 13, 14].

Abrial and Fraer used a B event-based approach to formalize Prim's algorithm in [6, 9]. A HOL-based formalization of graph search algorithms was done in [26]. Moore and Zhang verified Dijkstra's algorithm in ACL2[19]. Butler and Sjogren developed a graph library in PVS [7]. This library contains definitions for graphs, subgraphs, walks, paths, etc, as well as the proofs for Ramsay's and Menger's theorems. Nipkow et al. verified in HOL an enumeration of tame graphs as defined in Hales' proof of the Kepler Conjecture[24].

1.5 Organization of the Paper

Chapter 2 introduces the MIZAR Graph Library that is used as a base for our work. Chapter 3 is divided into four sections. The first introduces chordal graphs and discusses the proof of a characterization for this class of graphs. The second discusses a special class of graph algorithms we call vertex numbering algorithms. The third and fourth discuss our formalization of two chordal graph recognition algorithms: Lexicographic Breadth-First Search and Maximum Cardinality Search. Chapter 4 contains a summary of the thesis and our conclusion. We also offer four

appendices. Appendix A contains an introduction, Appendix B contains a section of Golumbic's chapter on chordal graphs, Appendix C contains our version of the same section edited to reflect the problems we encountered during our work, and Appendix D contains the MIZAR abstracts of our formalization.

Chapter 2

The MIZAR Graph Library

In this chapter we discuss the MIZAR Graph Library (abbreviated as GLIB from now on) created by Lee[12]. We highlight the definitions and theorems from GLIB that will be used throughout the remainder of this paper.

2.1 Graphs

A graph is a set with two elements, V and E. The set V contains the *vertices* of the graph and E contains the *edges* of the graph. The notation $G = (V, E)$ is commonly used to denote a graph G with vertex set V and edge set E. We require that V is not empty. If x and y are vertices of G, then we write the edge from x to y as xy. Graphs can be *directed* or *undirected*. In a directed graph every edge in E has an orientation – that is, we regard the edge as going from one vertex to another. The first vertex of each edge is called the *source* and the second vertex is called the *target*. We write xy for an edge from x to y and yx for an edge from y to x. Note that in a directed graph these edges are not the same. In an undirected graph edges are viewed without orientation, and so xy and yx are treated as the same. GLIB treats all graphs as directed graphs, but offers a simple mechanism to handle undirected graphs (Section 2.4 discusses this in more detail). A graph is *simple* if for each pair of vertices there is at most a single edge between them. Many of the results in this paper were first proved for simple graphs, but wherever possible we have done our formal proofs without this restriction.

Let $G = (V, E)$. We say the *order* of G is equal to $|V|$ and the *size* of G is equal to $|E|$. A graph is *finite* if both V and E are finite sets, that is, if its order and size are both finite. The MIZAR graph library was designed to handle both infinite and finite graphs, but we restrict our attention to finite graphs from now on.

A graph can be *labeled*. A *label* is a value associated with a vertex or an edge. An example of a graph labeling is a weight added to an edge that is used to denote distance, cost, or time. Another example is a flag set by an algorithm to mark that a vertex has been already processed. There are many uses for labels, but the main point is that labels are used to add extra information to a graph.

We will now discuss how GLIB defines a graph formally. Note that our goal here is to place as few restrictions as possible on our graphs.

Definition 2.1.1 *A* `GraphStruct` *is a function whose domain is a subset of the natural numbers.*

```
definition
  mode GraphStruct -> finite Function means :: GLIB_000:def 1
    dom it c= NAT;
end;
```

NOTE: this definition is from the MML article `glib_000.miz`, which is described in Lee[17]. All definitions cited from it will contain the GLIB_000 identifier as a comment in the definiton, like above. Similarly, definitions with the GLIB_001, GLIB_002, and GLIB_003 identifiers are described in Lee[18, 15, 16], respectively.

NOTE: the `mode` keyword is used to define an object that may not be unique. The keyword `it` refers to the object being defined. The functor `dom` is defined to return the domain of a function. The `c=` operator is the "subset of" operator, $\subseteq$.

Informally, a `GraphStruct` is any function defined on a subset of the natural numbers (`NAT` is the identifier used in the MML to refer to the set of naturals). Each element in the domain of a `GraphStruct` will be used to store a "part" of the graph. The elements of the domain are referred to from now on as *selectors*.

Any graph will require at least a set of vertices and a set of edges. All graphs in GLIB are directed by default and so each graph will also need to store information about which vertices are the sources and targets of each edge (how an undirected graph can be obtained is explained in Section 2.4). The selectors for these four properties are defined as follows:

```
definition
  func VertexSelector -> natural number equals  :: GLIB_000:def 2
    1;
  func EdgeSelector   -> natural number equals  :: GLIB_000:def 3
    2;
  func SourceSelector -> natural number equals  :: GLIB_000:def 4
    3;
  func TargetSelector -> natural number equals  :: GLIB_000:def 5
    4;
end;
```

Users of GLIB may add their own selectors as long as they do not conflict with those previously defined. The values used for a selector are completely arbitrary, of course. We show them here so the reader has an idea of how the underlying definitions are constructed.

More convenient and descriptive names for these selectors are defined:

```
definition let G be GraphStruct;
  func the_Vertices_of G equals  :: GLIB_000:def 7
    G.VertexSelector;
  func the_Edges_of G equals     :: GLIB_000:def 8
    G.EdgeSelector;
  func the_Source_of G equals    :: GLIB_000:def 9
    G.SourceSelector;
  func the_Target_of G equals    :: GLIB_000:def 10
    G.TargetSelector;
end;
```

To refer to the vertices in a graph G, for example, one can now write "`the_Vertices_of G`" instead of "`G.VertexSelector`". We found that this notation improves the readability of proofs written using GLIB, an important consideration when dealing with thousands of lines of text.

Up to this point we have defined only the underlying structure of a graph, so let us now define a graph.

Definition 2.1.2 *A* GraphStruct *G is* Graph-like *if* VertexSelector, EdgeSelector, SourceSelector, *and* TargetSelector *are in the domain of G,* the_Vertices_of G *is a non empty set, and* the_Source_of G *and* the_Target_of G *are both functions of* the_Edges_of G *and* the_Vertices_of G.

```
definition let G be GraphStruct;
  attr G is [Graph-like] means  :: GLIB_000:def 11
    VertexSelector in dom G & EdgeSelector in dom G &
    SourceSelector in dom G & TargetSelector in dom G &
    the_Vertices_of G is non empty set &
    the_Source_of G is Function of the_Edges_of G, the_Vertices_of G &
    the_Target_of G is Function of the_Edges_of G, the_Vertices_of G;
end;
```

A graph is then a [Graph-like] GraphStruct. A macro named _Graph is defined to make this shorter to write (the underscore is required because there is a previously defined graph structure in the MML that uses the name Graph).

We can test if two graphs are equal on these graph selectors by using the == predicate.

Definition 2.1.3 *G*1 == *G*2 *if and only if the vertices, edges, source function, and target function of G*1 *equal those of G*2.

2.2 Labeled Graphs

The advantage of having graphs defined in the above way is that we can add labels very easily. For example, GLIB contains three graph labelings:

```
definition
  func WeightSelector -> natural number equals  :: GLIB_003:def 1
    5; coherence;
  func ELabelSelector -> natural number equals  :: GLIB_003:def 2
    6; coherence;
  func VLabelSelector -> natural number equals  :: GLIB_003:def 3
    7; coherence;
end;

definition let G be GraphStruct;
  attr G is [Weighted] means                    :: GLIB_003:def 4
    WeightSelector in dom G &
    G.WeightSelector is ManySortedSet of the_Edges_of G;

  attr G is [ELabeled] means                    :: GLIB_003:def 4
    ELabelSelector in dom G &
    ex f being Function st G.ELabelSelector = f &
         dom f c= the_Edges_of G;

  attr G is [VLabeled] means                    :: GLIB_003:def 4
    VLabelSelector in dom G &
    ex f being Function st G.VLabelSelector = f &
         dom f c= the_Vertices_of G;
end;
```

These labelings are edge weights, edge labels, and vertex labels, respectively (a vertex weighting was not defined since it was not needed at the time GLIB was written). The difference between edge weights and edge labels is that the former is defined on all edges, and the later only on some subset of edges (a `ManySortedSet of I` is a function whose domain is exactly `I`). This means that in a `[Weighted]` graph every edge will have a weight, but in a `[ELabeled]` graph it is possible for there to be an edge with no label. No restriction is made on what these labelings map to. Each label is a function from the vertices or edges into some set.

A graph can have any combination of the above labels, and graphs with a subset of these labels can be referred to by specific names.

```
definition
  mode   WGraph is [Weighted]                          _Graph;
  mode   EGraph is             [ELabeled]              _Graph;
  mode   VGraph is                          [VLabeled] _Graph;
  mode  WEGraph is [Weighted] [ELabeled]               _Graph;
  mode  WVGraph is [Weighted]               [VLabeled] _Graph;
  mode  EVGraph is             [ELabeled] [VLabeled] _Graph;
  mode WEVGraph is [Weighted] [ELabeled] [VLabeled] _Graph;
end;
```

A `WEGraph` is a `_Graph` with edge weights and edge labelings, a `VGraph` is a graph with only a vertex label, and so on. MIZAR can determine the relationships between these types of graphs automatically since they are just attributes. This means it is not necessary to prove that a `WEGraph` is also a `WEVGraph`, for example.

Note that a labeling does not affect the truth of the == predicate defined above. That is, == is true only if the graphs are equal on the four base selectors, whereas = is true only if the graphs are exactly the same on all selectors. For example, let $G1$ and $G2$ be the same graph except that $G2$ contains a vertex label on some vertex that $G1$ does not. In this case, $G1 == G2$ is true because the vertex, edge, source, and target functions are all the same. But it is not true that $G1 = G2$ since $G2$ contains an extra label that $G1$ does not. Consequently, $G1 = G2$ implies $G1 == G2$ but $G1 == G2$ does not imply $G1 = G2$. It is easy and convenient to informally consider a graph with a label as the same graph as without the label—indeed, this is the purpose behind having the == predicate—but, be aware that this is not the case formally.

For the rest of this paper, we use only the vertex labeling property defined above. The edge labels and weights are not used since they were not needed by the algorithms we will be working with.

2.3 Subgraphs

A subgraph of a graph is defined as follows:

Definition 2.3.1 *Let $G = (V_g, E_g)$ and $H = (V_h, E_h)$ be graphs. H is a subgraph of G if $V_h \subseteq V_g$ and $E_h \subseteq E_g$.*

```
definition let G be _Graph;
  mode Subgraph of G -> _Graph means          :: GLIB_000:def 34
   the_Vertices_of it c= the_Vertices_of G &
    the_Edges_of it c= the_Edges_of G &
```

```
    for e being set st e in the_Edges_of it holds
      (the_Source_of it).e = (the_Source_of G).e &
      (the_Target_of it).e = (the_Target_of G).e;
end;
```

Notice that no mention is made of any extra labelings that may be present in the subgraph or the original graph. For example, it would be possible for H to be a subgraph of G and for H to have a vertex labeling defined on its vertices that G lacks. The type of subgraph defined above is strictly on the initial four graph selectors.

The notion of label-inheriting subgraphs was introduced to overcome this limitation.

```
definition let G be VGraph, G2 be [VLabeled] Subgraph of G;
  attr G2 is vlabel-inheriting means          :: GLIB_003:def 12
    the_VLabel_of  G2 = (the_VLabel_of G) | the_Vertices_of G2;
end;
```

The vertex label on the subgraph is the restriction of the original graph's vertex label to the subgraph's vertices (the | operator restricts the function on the left-hand side to the domain on the right-hand side).

It is necessary to introduce these types of label-inheriting subgraphs explicitly for every graph labeling defined.

One final type of subgraph that is used frequently in this paper is the subgraph induced by a set of vertices.

Definition 2.3.2 *Let $G = (V, E)$ be a graph, S be a non-empty subset of V. The subgraph of G induced by S is the graph $H = (S, F)$ where F is the set of edges with both endpoints in S.*

```
definition let G be _Graph, V, E be set;
 mode inducedSubgraph of G,V,E -> Subgraph of G means :: GLIB_000:def 39
   the_Vertices_of it = V & the_Edges_of it = E if
      V is non empty Subset of the_Vertices_of G &
      E c= G.edgesBetween(V)
    otherwise it == G;
```

The `G.edgesBetween(V)` function returns those edges whose endpoints both lie in V. Notice that the MIZAR version allows a subgraph to be induced on a set of vertices and edges; to obtain a subgraph induced only on the vertices there is a macro with the same name that accepts only the set V and substitutes `G.edgesBetween(V)` for E automatically. The latter version is the most commonly used.

Note that this definition does not guarantee a unique subgraph. While it is true there is only a single subgraph induced by any pair of V and E when considering only the four original graph selectors, there are an infinite number of graphs induced by V and E when we take into account any graph labelings present in the graph.

There are also label-inheriting versions of `inducedSubgraph` that are defined similarly to the above.

2.4 Connectivity

We can talk about edge connectivity via the Joins predicate.

Definition 2.4.1 *The edge* e Joins *the two vertices* x *and* y *in* G *if one is the source and the other is the target of* e.

```
definition let G be _Graph, x,y,e be set;
   pred e Joins x,y,G means                    :: GLIB_001:def 1
     e in the_Edges_of G &
     (((the_Source_of G).e = x & (the_Target_of G).e = y) or
      ((the_Source_of G).e = y & (the_Target_of G).e = x));
end;
```

Previously we mentioned that all graphs in GLIB are directed. It is in using the Joins predicate that we can consider a graph to be undirected, since edge orientation is ignored.

2.5 Walks and Paths

Various texts differ in what they mean by a walk, a trail, and a path. The GLIB definitions of these notions are as follows; for a more in depth discussion about these definitions see Lee[12].

Definition 2.5.1 *A* Walk *is any odd length alternating sequence of vertices and edges with the property that every edge in the sequence joins the preceeding and succeeding vertices.*

```
definition let G be _Graph;
  mode Walk of G -> FinSequence of the_Vertices_of G \/
                    the_Edges_of G means        :: GLIB_001:def 3
    len it is odd & it.1 in the_Vertices_of G &
    for n being odd natural number st n < len it
      holds it.(n+1) Joins it.n, it.(n+2), G;
end;
```

Notice that a single vertex is a Walk, all odd indices in a walk map to vertices, and all even indices map to edges.

Definition 2.5.2 *A walk is* Trail-like *if no edge is repeated.*

Definition 2.5.3 *A walk is* Path-like *if it is* Trail-like *and if no vertices are repeated, except possibly the first and last vertices.*

```
definition let G be _Graph, W be Walk of G;
  attr W is Path-like means                    :: GLIB_001:def 28
    W is Trail-like &
    for m, n being odd natural number st m < n & n <= len W holds
      W.m = W.n implies (m = 1 & n = len W);
end;
```

A Walk is closed if its first and last vertices are equal and is open otherwise. A walk with a single vertex is considered trivial; otherwise it is non-trivial.

Definition 2.5.4 *A walk is* Cycle-like *if it is* closed *and* Path-like *and* non-trivial.

2.6 Graph Sequences/Algorithms

In GLIB, a graph algorithm is defined as an infinite sequence of graphs, each of which corresponds to a single "step" of the algorithm. A typical step could include adding a label to a vertex or changing the edge weights on a set of edges. All information the algorithm requires must be stored as labelings – there is no support for outside queues or stacks. Any such data structure must be simulated with a graph labeling of some kind.

Because of these constraints, we have not considered talking about the true time complexity of an algorithm in the traditional sense. We may state that a particular algorithm will halt after a certain number of graph operations, but it is usually difficult to say more.

We consider a sequence of graphs to be **halting** if there is some natural n such that the nth and $(n+1)$st graphs of the sequence are equal — this is true equality, not equality according to the == predicate (Definition 2.1.3).

Definition 2.6.1 *A Graph Sequence GS is* halting *if there exists a natural number n such that* $GS.n = GS.(n+1)$.

The first index where subsequent graphs in the sequence are the same is called the *lifespan* of the sequence.

Definition 2.6.2 *Let GS be a graph sequence. The* lifespan *of GS equals n where n is the smallest natural so that* $GS.n = GS.(n+1)$ *if GS is* `halting` *and* $n = 0$ *otherwise.*

```
definition let F be ManySortedSet of NAT;
  func F.Lifespan() -> natural number means       :: GLIB_000:def 57
    F.it = F.(it+1) & for n being natural number
      st F.n = F.(n+1) holds it <= n
    if F is halting otherwise it = 0;
end;
```

Since we will be dealing with finite graphs, it is useful to consider graph sequences that are composed of finite graphs.

Definition 2.6.3 *A Graph Sequence GS is* finite *if for each natural n it is true that GS.n is a finite graph.*

This definition can be a little confusing since when used it looks like the graph sequence itself is finite (i.e. contains a finite number of graphs), however, this is not the case. Perhaps a better name for this attribute would have been *finite-yielding*.

GLIB contains the proofs of correctness for three well-known graph algorithms using the above graph sequences: Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree algorithm, and Ford-Fulkerson's Maxflow algorithm. See Lee[12] for the proofs of correctness for these algorithms.

Chapter 3

Formalized Proofs

We prove a characterization for the class of chordal graphs and define and verify the correctness of two recognition algorithms. In our developments we follow closely the chapter on triangulated graphs in Golumbic[10]. See Appendices B and C for the original and revised versions of this chapter.

3.1 Characterization of Chordal Graphs

In this section we add some definitions to the graph library in order to work with the class of chordal graphs. We then prove a characterization for this class of graphs. All MIZAR code snippets in this section are from the file listed in D.1.

3.1.1 Definitions

We first discuss some miscellaneous definitions we found useful and then proceed with the definitions leading up to the class of chordal graphs.

Definition 3.1.1 *Let* $v1$ *and* $v2$ *be vertices of* G. *The predicate* $v1, v2$ `are_adjacent` *holds if there is an edge* e *that* `Joins` $v1$ *and* $v2$ *in* G.

```
definition let G be _Graph, a,b be Vertex of G;
  pred a,b are_adjacent means                    :: CHORD:def 3
    ex e being set st e Joins a,b,G;
end;
```

This definition is redundant as it is only a wrapper for the `Joins` predicate; however, using it results in theorems with nicer statements than those that use the `Joins` predicate.

Definition 3.1.2 *Given a set* S *and a graph* G, `G.AdjacentSet(S)` *returns the subset of the vertices not in* S *such that each element of this set is adjacent to some element of* S.

```
definition let G be _Graph, S be set;
  func G.AdjacentSet(S) -> Subset of the_Vertices_of G equals
:: CHORD:def 4
   {u where u is Vertex of G :
      not u in S & ex v being Vertex of G
       st (v in S & u,v are_adjacent)};
end;
```

Definition 3.1.3 *A subgraph of G is an* AdjGraph *of G and a set S, if it equals the subgraph of G induced by* G.AdjacentSet(S).

```
definition let G be _Graph, S be set;
  mode AdjGraph of G,S -> Subgraph of G means     :: CHORD:def 5
    it is inducedSubgraph of G,G.AdjacentSet(S)
    if S is Subset of the_Vertices_of G;
end;
```

This notion is used only in the definition of a *simplicial* vertex (see Def 3.1.7). We added a minlength attribute for walks in finite graphs:

Definition 3.1.4 *Let W be a walk from v to w in a graph G. Then W is* minlength *if every walk from v to w is at least as long as W.*

```
definition let G be finite _Graph, W be Walk of G;
  attr W is minlength means                       :: CHORD:def 2
    for W2 being Walk of G st W2 is_Walk_from W.first(),W.last()
      holds len W2 >= len W;
end;
```

Definition 3.1.5 [10, p.82] *A subset S of V is a* vertex separator *of non adjacent vertices a and b if the removal of S from G separates a and b into different components.*

```
definition let G be _Graph, a,b be Vertex of G;
 assume A0: a<>b & not a,b are_adjacent;
 mode VertexSeparator of a,b ->
      Subset of the_Vertices_of G means          :: CHORD:def 8
   not a in it & not b in it &
   for G2 being removeVertices of G,it holds
      not (ex W being Walk of G2 st W is_Walk_from a,b);
```

We say that S is a *minimal* vertex separator of a and b if no proper subset of S is a vertex separator of a and b. If a and b are in different components of the graph, then any subset of $V - \{a, b\}$ is trivially a vertex separator of a and b and the empty set is a minimal separator of a and b. Note that in a finite graph there may be many minimal vertex separators for a pair of vertices that are distinct and non adjacent (i.e. not connected by an edge), but that there is always at least one.

Definition 3.1.6 [10, p.5] *A graph is* complete *if every pair of distinct vertices is joined by an edge.*

```
definition let G be _Graph;
  attr G is complete means                        :: CHORD:def 6
    for u,v being Vertex of G st u <> v holds u,v are_adjacent;
end;
```

Notice this definition is concerned only with distinct vertices and so we are making no assumptions about loops (edges with the same source and target).

Definition 3.1.7 [10, p.82] *A vertex v is* simplicial *if the subgraph induced by the neighbors of v is a complete graph.*

```
definition let G be _Graph, v be Vertex of G;
  attr v is simplicial means                    : DefSimplicial :
   G.AdjacentSet({v}) <> {} implies
       for G2 being AdjGraph of G,{v} holds G2 is complete;
end;
```

NOTE: the symbol <> stands for "not equals."

Recall that we have previously defined `G.AdjacentSet(S)` as the vertices of `G` that are adjacent to some member of `S` (Definition 3.1.2) and `AdjGraph of G,S` as a subgraph of `G` induced by `G.AdjacentSet(S)` (Definition 3.1.3). Note that we have chosen to include isolated vertices as simplicial vertices and that these are handled by a special case that arises because we cannot consider graphs with an empty vertex set.

The algorithms we will be working with require the notion of a *vertex scheme* (sometimes referred to as a *vertex order*), which is a total ordering of the vertices.

Definition 3.1.8 *A* `VertexScheme` *of a graph G is an ordering of the vertices of G.*

```
definition let G be finite _Graph;
  mode VertexScheme of G -> FinSequence of the_Vertices_of G means
:: CHORD:def 12
    it is one-to-one & rng it = the_Vertices_of G;
end;
```

Note: `rng it` refers to the range of finite sequences we are defining.

The function `followSet(n)` on a `VertexScheme` returns the set of vertices occurring after the nth index of the ordering. This allows us to define:

Definition 3.1.9 [10, p.5] *An ordering of the vertices $\sigma = [v_1, \ldots, v_n]$ is a* perfect vertex elimination scheme *if for each $1 \leq i \leq n$ the vertex v_i is simplicial in the subgraph induced by the vertices $\{v_i, v_{i+1}, \ldots, v_n\}$.*

```
definition let G be finite _Graph, S be VertexScheme of G;
  attr S is perfect means                          :: CHORD:def 13
    for n being non empty natural number st n <= len S
    for Gf being inducedSubgraph of G,S.followSet(n)
    for v being Vertex of Gf st v = S.n holds v is simplicial;
end;
```

To define the class of chordal graphs, we first introduce the notion of a chordal walk.

Definition 3.1.10 *Let w be a walk in an undirected graph $G = (V, E)$. An edge $e \in E$ is said to be a* chord *of w if e is not in the edges of w and e joins two distinct non-consecutive vertices of w.*

```
definition let G be _Graph, W be Walk of G;
  attr W is chordal means                          :: CHORD:def 10
  ex m, n being odd natural number st
    m+2 < n & n <= len W & W.m <> W.n &
    (ex e being set st e Joins W.m,W.n,G) &
    for f being set st f in W.edges() holds not f Joins W.m,W.n,G;
end;
```

We consider w to be *chordal* if there exists a chord of w and *chordless* if there is no chord in w.

Recall that a walk of G (Definition 2.5.1) is an alternating sequence of vertices and edges of G. This is why the indices of the endpoints of the chord (here m and n) are odd and n is not the next largest odd index (i.e. $m+1$). Note this definition disallows loops as chords. Since a cycle is a non trivial closed path we may also consider chordal and chordless cycles.

When using the above definition on cycles we found it was convenient to consider a slightly different and more technical version:

```
theorem ChordalWalk01:                                    :: CHORD:84
for G being _Graph, W being Walk of G st W is chordal
  ex m,n being odd natural number st
     m+2 < n & n <= len W & W.m <> W.n &
     (ex e being set st e Joins W.m,W.n,G) &
     (W is Cycle-like implies  not (m=1 & n = len W) &
                               not (m=1 & n = len W-2) &
                               not (m=3 & n = len W))
```

This version explicitly disallows three edges that become a problem when working with cycles (to see why these edges are a problem, notice that they should not be regarded as chords since they join subsequent vertices of the cycle, but that the numerical value of their indices pass the $m+2<n$ condition).

A seemingly simple notion like a chord is actually quite tricky to state formally in an accurate and useful way. Sometimes we must accept that the formal definition will be ugly for it to be useful.

We are finally in a position to define the class of chordal graphs.

Definition 3.1.11 [10, p.81] *A graph G is chordal if every cycle of length greater than three is chordal.*

```
definition let G be _Graph;
 attr G is chordal means                    :: CHORD:def 11
   for P being Walk of G st P.length() > 3 & P is Cycle-like
     holds P is chordal;
end;
```

We are now in a position to prove a characterization of this class of graphs.

3.1.2 Characterization

We wish to verify the proof of the following classification theorem for chordal graphs from Golumbic[10].

Theorem 3.1.12 [10, p.83] *Let G be an undirected graph. The following statements are equivalent:*

(i) G is chordal.

(ii) G has a perfect vertex elimination scheme. Moreover, any *simplicial vertex can start a perfect scheme.*

(iii) Every minimal vertex separator induces a complete subgraph of G.

The theorem as stated above makes two hidden assumptions. First, G is finite. This is fine, and we have already stated we are dealing exclusively with finite graphs. Secondly, G is connected. If G were disconnected, then two vertices in separate components would have an empty minimal vertex separator. This is fine if we assume the empty graph is complete. In GLIB however, we have no way to deal with empty graphs so we need to avoid this case by assuming G is connected.

We should mention that MIZAR does not support three-way equivalences like we have in the statement of Theorem 3.1.12, so we treat each implication separately.

Let us first discuss (i) $\Rightarrow$ (iii). The statement of the theorem in MIZAR is as follows.

```
theorem Chordal41:  :: Chordal41
for G being chordal _Graph,
    a,b being Vertex of G
 st a<>b & not a,b are_adjacent
    for S being VertexSeparator of a,b
      st S is minimal & S is non empty
        for H being inducedSubgraph of G,S holds H is complete
```

This statement is longer than the textbook version for two reasons. The first is that when we refer to a vertex separator we must mention which vertices it separates. In order to state the theorem precisely we need to talk about all viable pairs of vertices that can have vertex separators instead of just the separators themselves, like the textbook version does. We also need to add the clause "S is non empty" to avoid the problem of an empty set inducing a subgraph. Note also that we cannot talk about *the* induced subgraph of S, but instead need to talk about *any* induced subgraph of S. This is because of the way graphs are defined in the MIZAR graph library. In particular, vertex and edge labelings on these graphs give rise to a number of subgraphs induced by S, each with the same underlying graph structure, but different nonetheless.

The textbook version of the proof is straightforward and proceeds as follows (this is a verbatim copy from Golumbic[10]):

Proof. (i) $\Rightarrow$ (iii) Suppose S is a minimal $a-b$ separator with G_A and G_B being the connected components of G_{V-S} containing a and b, respectively. Since S is minimal, each $x \in S$ is adjacent to some vertex in A and some vertex in B. Therefore, for any pair $x, y \in S$ there exist paths $[x, a_1, \ldots, a_r, y]$ and $[y, b_1, \ldots, b_t, x]$, where each $a_i \in A$ and $b_i \in B$, such that these paths are chosen to be of smallest possible length. It follows that $[x, a_1, \ldots, a_r, y, b_1, \ldots, b_t, x]$ is a simple cycle whose length is at least 4, implying that it must have a chord. But $a_i b_j \notin E$ by the definition of vertex separator, and $a_i a_j \notin E$ and $b_i b_j \notin E$ by the minimality of r and t. ∎

It is worthwhile discussing a few aspects of how this proof was done in MIZAR to illustrate some of the difficulties in writing graph theoretic proofs formally. The proof above is followed almost verbatim in the formal proof; it is only how the intermediate results are obtained that is interesting here.

One thing to mention is that the proof is a little stronger than it needs to be. That is, we do not need to obtain the shortest paths connecting x and y through G_A and G_B, we need only obtain two *chordless* paths connecting x and y through G_A and G_B (any shortest path between two vertices will necessarily be chordless).

It is this chordless property that we need to finish the proof. Either way, it is trivial to obtain a shortest (or chordless) path between two vertices in a finite connected graph.

In the MIZAR proof, we first obtain the subgraph induced by G_{V-S} and then the connected components G_A and G_B of G_{V-S}. We consider x and y in G_S and need to show they are adjacent. Now comes a slight problem: how do we obtain the paths between x and y going through G_A in a nice fashion? We cannot just consider x and y as vertices of G and ask for the shortest path between them, because this would not guarantee the path going through G_A or G_B. The first solution we tried, which follows the book, was to consider the points a_x and a_y adjacent to x and y in G_A (this itself is a tricky little result to do nicely) and the shortest path between a_x and a_y in G_A. We do the same with G_B. To construct the cycle needed in the proof, it is then necessary to add the edge between a_y and y to the path in G_A, then the edge between y and b_y, then the path from b_y to b_x in G_B, then the edge from b_x to x, etc. At each step it is necessary to show the concatenated walk is still a path and that it is still chordless (this can get quite tedious and lengthy). This approach works but is quite ugly and long. A better way to do it is to simply construct the subgraphs induced by $G_A \cup \{x, y\}$ and $G_B \cup \{x, y\}$, show they are both connected subgraphs, then take the shortest path between x and y in both of these graphs. This leaves us with two paths that we concatenate together to obtain the cycle. This approach is much shorter and cleaner than the first since we get most of what we need for "free".

Notice how the textbook proof implicitly implies the first approach by stating the existence of a_x and a_y and how G_A is connected. No mention is made of which graph this path comes from; in particular, there is no mention of the graphs $G_A \cup \{x, y\}$ and $G_B \cup \{x, y\}$. A proof that works well on paper may be very cumbersome and long when done formally; in such cases it is often necessary to do the proof in a fashion very different from the paper version.

The proof of (iii) $\Rightarrow$ (i) is straightforward and there is little difficulty in writing its formalized version. For completeness, the MIZAR statement of it is given here.

```
theorem :: DiracThm2:
for G being finite _Graph st
 for a,b being Vertex of G st a<>b & not a,b are_adjacent holds
     for S being VertexSeparator of a,b
        st S is minimal & S is non empty
     for G2 being inducedSubgraph of G,S
        holds G2 is complete
 holds G is chordal
```

The next step of the proof of Theorem 3.1.12 is to show the equivalence between (i) and (ii). The MIZAR statement for (i) $\Rightarrow$ (ii) is:

```
theorem
  for G being finite chordal _Graph holds
    ex S being VertexScheme of G st S is perfect;
end;
```

The statement of (ii) $\Rightarrow$ (i) is written as:

```
theorem :: Chordal41c:
```

```
for G being finite _Graph holds
   (ex S being VertexScheme of G st S is perfect)
      implies G is chordal
```

Both of these proofs follow verbatim the proofs in Golumbic[10] with little difficulty.

There is one final thing to mention. In the proof of Lemma 4.2 in Golumbic[10, p.83] we prove there is a simplicial vertex of G in the connected component A. The textbook says, "Similarly B contains a simplicial vertex of G". That is, the proof of the existence of a simplicial vertex in component B is almost the same, except for the names A and B, as the proof in component A. This is fine on paper, but to do this in MIZAR we were required to replicate the entire section verbatim except for the name changes.

3.1.3 Discussion

In order to prove Theorem 3.1.12 around 5000 lines of background definitions and facts needed to be proved. The proof of Theorem 3.1.12 itself is an additional 1000 lines. There were no major obstacles to be overcome in formalizing this result. Most of the machinery we needed was already present, and the facts that were not already present were not difficult to add.

As a further test of the robustness of our formalization, we verified the proof of two exercises from the section in Golubmic[10]. Both exercises deal with minimal vertex separators. Let S be a minimal vertex separator of x and y. The first problem asks to show that every path between x and y contains some element of S, and that every element of S appears in some path between x and y as the lone element of S in the path. The second problem asks to show that there is always a vertex in a connected component of $G - S$ adjacent to all of S. Both results were proved with a modest amount of effort. All the machinery required to do the proofs was available for us; that is, no extra formal machinery needed to be added in order to complete these proofs.

3.2 Vertex Numbering Graph Sequences

We now introduce a class of graph sequences that we call `vlabel-numbering` sequences (the name refers to the MIZAR implementation in which we number each vertex via the vertex-label of GLIB). We define such sequences since they possess several properties that both LexBFS and MCS algorithms require.

Before we do so, however, we need to add a few definitions dealing with graph sequences. This is because the original halting attribute (Definition 2.6.1) is not very useful, and the `Lifespan` function (Definition 2.6.2) is not well defined on non-halting algorithms. To see this, consider two sequences: a constant sequence and a random non-halting sequence. The first has a lifespan of 0 and the second also has a lifespan of 0.

To overcome this problem, we introduced the notions of *iterative* and *eventually-constant* sequences.

Definition 3.2.1 *A sequence is* iterative *if whenever elements n and k are equal it holds that elements $n+1$ and $k+1$ are also equal.*

```
definition let s be ManySortedSet of NAT;
  attr s is iterative means
:: LEXBFS:def 15
    for k, n being natural number st s.k = s.n holds s.(k+1) = s.(n+1);
end;
```

This property is asserting that the sequence behaves in a deterministic fashion — that is, a state always leads to the same successor state.

Definition 3.2.2 *A graph sequence is* eventually-constant *if there is a natural number n such that for all $m \geq n$ it holds that elements n and m are equal.*

```
definition let GS be ManySortedSet of NAT;
  attr GS is eventually-constant means
:: LEXBFS:def 16
    ex n being natural number
     st for m being natural number st n <= m holds GS.n = GS.m;
end;
```

With these two definitions we can easily prove in MIZAR the following three theorems for a graph sequence S.

Theorem 3.2.3 *If S is* halting *and* iterative *then S is* eventually-constant.

Theorem 3.2.4 *If S is* eventually-constant *then S is* halting.

Theorem 3.2.5 *If S is* iterative *and* eventually-constant, *then all elements after* `S.Lifespan()` *are equal (i.e. the natural number in the Def 3.2.2 is the same as* `S.Lifespan()`*).*

We are now in a position to define a `vlabel-numbering` sequence.

Definition 3.2.6 *A graph sequence is* vlabel-numbering *if it is iterative, eventually-constant, composed of finite graphs whose vertices and edges are static throughout the sequence, has a vertex-labeling that is empty to start, and with each step numbers exactly one unnumbered vertex with a natural number that decreases by one on each step.*

```
definition let GS be VGraphSeq;
  attr GS is vlabel-numbering means
:: LEXBFS:def 26
  GS is iterative
        eventually-constant
        finite
        fixed-vertices
        natural-vlabeled
        vlabel-initially-empty
        adds-one-at-a-step;
end;
```

A `VGraphSeq` is a sequence of `VGraphs`, that is, graphs with vertex labels. The `finite` property is the same as that of Definition 2.6.3. The `fixed-vertices` property means that every graph of the sequence is equal modulo any labelings, that is, the vertices remain the same from step to step. The `natural-vlabeled`

property means that the vertex-label of each graph in the sequence maps to the natural numbers. The `adds-one-at-a-step` property ensures that at each step n until the algorithm halts, there is some unnumbered vertex w that is numbered with $|V| - n$ and that the vertex-labeling of the subsequent step reflects this update.

There are a few useful properties of graph sequences with the `vlabel-numbering` property that we prove here. Let $G = (V, E)$ be the graph the sequences below are defined on, so that $|V|$ in the following lemmas refers to the number of vertices in this graph (or to any graph of the sequence since they must all have the same vertex set). Note that when we mention a step n of a sequence we make the implicit assumption that n occurs before the sequence halts, that is, $n \leq |V|$. We will denote the label of the nth step in the sequence by σ_n. We write $\sigma_n(v) = k$ to mean that in the nth step of the algorithm, vertex v is labeled with k. We use $[a, b]$ to denote the set of values from a to b, inclusive.

Lemma 3.2.7 *The total number of vertices that have been numbered by the end of the nth step of a* vlabel-numbering *graph sequence is n.*

```
theorem:
  for GS being vlabel-numbering VGraphSeq, n being natural number
    st n <= GS.Lifespan() holds card ((GS.n).labeledV()) = n
```

Proof. We proceed by induction on n for $n \leq GS.Lifespan()$. If $n = 0$ then $\sigma_n = \emptyset$ by Definition 3.2.6, and so the base case holds. Suppose that n vertices have been numbered after step n. Again by Definition 3.2.6, we know there is a vertex w not in the domain of σ_n, but in the domain of σ_{n+1}, so the inductive step holds. ∎

Lemma 3.2.8 *The range of numbers assigned to the first n numbered vertices of a* vlabel-numbering *graph sequence is* $[|V| - n + 1, |V|]$.

```
theorem :: LEXBFS:16
for GS being vlabel-numbering VGraphSeq, n being natural number
  holds rng the_VLabel_of (GS.n) =
        (Seg GS.Lifespan()) \ Seg (GS.Lifespan()-'n)
```

Note: the MIZAR function **Seg n** returns the set $[1, n]$ and the backslash character denotes set subtraction.

Proof. We proceed by induction on n for $n \leq GS.Lifespan()$. If $n = 0$ then the base case holds by Definition 3.2.6. Suppose that the range of numbers assigned to the n numbered vertices after step n is $[|V| - n + 1, |V|]$. By Definition 3.2.6, there is a vertex w with number $|V| - n$ at step $n + 1$, and so the range of values at step $n + 1$ is $[|V| - n + 1, |V|] \cup \{|V| - n\} = [|V| - n, |V|] = [|V| - (n + 1) - 1, |V|]$. ∎

Lemma 3.2.9 *The labeling function of the nth step of a* vlabel-numbering *graph sequence is one-to-one.*

```
theorem :: LEXBFS:20
for GS being vlabel-numbering VGraphSeq, n being natural number holds
  the_VLabel_of (GS.n) is one-to-one
```

Proof. This follows directly from Definitions 3.2.7 and 3.2.8. ∎

Lemma 3.2.10 *Let m and n be so that $m \leq n$. Then $\sigma_m \subset \sigma_n$.*

```
theorem :: LEXBFS:19
for GS being vlabel-numbering VGraphSeq,
    m,n being natural number st m <= n
  holds the_VLabel_of (GS.m) c= the_VLabel_of (GS.n)
```

Proof. We proceed by induction on k. For $k = 0$, it is trivially true that $\sigma_m \subseteq \sigma_{m+k}$. Suppose for some k that $\sigma_m \subset \sigma_{m+k}$. Let w be the vertex numbered in step $m+k$. Then w is not in the domain of σ_{m+k}, but must be in the domain of $\sigma_{m+(k+1)}$ by Definition 3.2.6. All other labeled vertices of step $m+k$ are also in the label of step $m+(k+1)$, and so $\sigma_m \subset \sigma_{m+(k+1)}$. ∎

Lemma 3.2.11 *A vertex with number n is chosen at step $|V| - n$.*

```
theorem :: LEXBFS:22
for GS being vlabel-numbering VGraphSeq, v being set,
    m,n being natural number
 st (v in (GS.m).labeledV() & (the_VLabel_of (GS.m)).v = n)
  holds GS.PickedAt(GS.Lifespan()-'n) = v
```

Note: the function `PickedAt(n)` returns the vertex chosen at step n.

Proof. Let v have the number n at some point late in the algorithm (say at step m) and let w be the vertex chosen at step $j = |V| - n$. We need to show that $v = w$. We know that $j < m$, since v would not have a number otherwise. Since w was chosen in step j, it is in the domain of σ_{j+1} and has number $|V| - (|V| - n) = n$, and so $\sigma_{j+1}(w) = n$. We know that $\sigma_{j+1} \subset \sigma_m$ by Lemma 3.2.10, so w is also in the domain of σ_m. Since $\sigma_m(w) = n = \sigma_m(v)$ and σ_m is one-to-one by Lemma 3.2.9, it follows that $v = w$. ∎

Lemma 3.2.12 *Let a and b be vertices such that b has a larger vertex label than a at some point in the algorithm, then b was in the vertex label of the step in which a was chosen to be numbered.*

```
theorem :: LEXBFS:25
for GS being vlabel-numbering VGraphSeq
for i being natural number, a,b being set
 st a in (GS.i).labeledV() & b in (GS.i).labeledV() &
    (the_VLabel_of (GS.i)).a < (the_VLabel_of (GS.i)).b
  holds b in (GS.(GS.Lifespan() -'
            (the_VLabel_of (GS.i)).a)).labeledV()
```

Proof. Let a and b be numbered vertices of step i, and let n_a and n_b be such that $\sigma_i(a) = n_a$ and $\sigma_i(b) = n_b$. Then a was chosen in step $|V| - n_a$ and b was chosen in step $|V| - n_b$ by Lemma 3.2.11. Since $n_a < n_b$, we know that $|V| - n_b < |V| - n_a$. Then b is in the domain of $\sigma_{|V|-n_b+1}$ by Definition 3.2.6 and so is also in the domain of $\sigma_{|V|-n_a}$ by Lemma 3.2.9. ∎

There are several other small facts that we found useful but we omit them here for brevity. The examples above should give an idea of the type of facts we will use later and the nature of their proofs. To see all required facts see Appendix D.2.

3.3 Lexicographical Breadth-first Search Algorithm

3.3.1 Recognizing Chordal Graphs

We are interested in determining whether a given finite undirected graph belongs to the class of graphs called chordal graphs. Recall that a chordal graph is a graph which contains no chordless cycle of length greater than three.

In section 3.1 we showed that every chordal graph contains a simplicial vertex (Theorem 3.1.12). If a vertex is simplicial then it cannot be on a chordless cycle with length greater than three. These two facts combined imply that if v is a simplicial vertex of G then G is chordal if and only if $G - v$ is chordal. If we repeatedly find and remove a simplicial vertex until the graph has fewer than four vertices then we have proven the graph chordal. The order in which we remove the simplicial vertices is then a perfect vertex elimination scheme. Theorem 3.1.12 guarantees that every chordal graph has such an order and that any graph with such an order is chordal. Also notice that it does not matter which simplicial vertex we start with, so in order to verify that a graph is chordal we need only find any simplicial vertex — we do not have to try all orders of removal.

We can check if a vertex is simplicial in $O(|V|^2)$ time (simply check that all pairs of neighbors are adjacent). We can then determine if a graph has a simplicial vertex in $O(|V|^3)$ time, and so we can check if a graph is chordal in $O(|V|^4)$ time. It is possible to do much better than this, however.

In the following section we discuss a linear time algorithm for chordal graph recognition.

3.3.2 The LexBFS algorithm

In this section we discuss a linear time algorithm for recognizing chordal graphs, known as lexicographic breadth-first search.

The version of LexBFS that we translate and verify in MIZAR is found in Golumbic[10] (see Algorithm 1 for pseudo-code). LexBFS takes any finite undirected graph $G = (V, E)$ as input and returns an ordering σ of the vertices of G. We need to prove σ is a perfect vertex elimination order if G is chordal; the other direction is handled by Theorem 3.1.12.

The algorithm itself is quite simple. Each vertex is assigned a label that is initially empty; this label defines an ordering of the vertices that is used to determine which frontier node to explore at each step of the algorithm. In addition to this label, each vertex is also assigned a number when it is chosen. Vertices that have not been chosen are said to be *unnumbered*, vertices that have been chosen *numbered*. We say vertex v is numbered with n when $\sigma(n) = v$, i.e. v is the nth vertex in the ordering. Once a vertex is numbered it is never considered for numbering again. The number a vertex receives is equal to $|V| - i$, where i is the number of vertices numbered before it. Notice that the assigned numbers decrease from $|V|$ to 1, so a vertex with a larger number than another was chosen *before*.

The label can be implemented as an array of integers and these arrays are ordered with the lexicographic ordering. That is, label L_1 is said to be greater than label L_2 only if there is some index j such that for all positions $i < j$ it is true that $L_1(i) = L_2(i)$ and $L_1(j) > L_2(j)$. In our case, when we add the number i to a label in step 4 of the algorithm we are appending the new number to the end of the array

Algorithm 1 Lexicographic BFS

Require: Finite undirected graph $G = (V, E)$
Ensure: Ordering σ of V

```
Assign the label ∅ to each vertex
for i = |V| downto 1 do
    v ← any unnumbered vertex with largest label
    σ(i) = v
    for all unnumbered vertices w ∈ Adj(v) do
        Add i to w's label
    end for
end for
return σ
```

(label). It so happens that this number will be smaller than all numbers currently in the array (label).

How this labeling is implemented in practice will affect the runtime complexity of the algorithm, since this will affect how the vertices are compared and stored in an ordered list. A true $O(|V| + |E|)$ worst-case runtime can be achieved by storing the labeling implicitly, and for completeness sake we outline this idea below (a similar idea is found on page 87 of Golumbic[10]).

We initialize a linked list of sets Q to be a single entry. This linked list will contain subsets of the vertices that have the same label, that is, those vertices that are equal with respect to the ordering. The first entry of Q contains the vertices with highest ranking; later sets in Q contain vertices of lower ranking. At first, the only set in Q contains all the vertices of the graph G. When we wish to number a vertex, we choose any vertex x from the first set of Q and remove it from this set. We now need to update the "label" on the neighbors of x. Any set T in Q that contains a vertex adjacent to x is split into two sets T_1 and T_2, where T_1 contains those members of T adjacent to x and T_2 contains those not adjacent to x. In Q, T_1 comes directly before T_2. This can be done so that for each vertex we do work proportional to the number of edges adjacent to it, so the total complexity is indeed $O(|V| + |E|)$, since we visit each vertex and each edge once.

From the way graph algorithms are handled in the MIZAR graph library, we have no easy way to prove that the complexity of this algorithm is $O(|V| + |E|)$, so the above discussion does not affect us here. In particular, at the current state of the formalization we have no way of stating how much work it takes to choose a vertex, update its neighbors, and re-sort the unnumbered vertices at each step. The most we can show is that the algorithm will terminate after exactly $|V|$ graph operations; or simply, after each vertex has been numbered.

In the next section we discuss the formulation of the algorithm in MIZAR.

3.3.3 Formal Definitions of LexBFS

We will use the vertex-labeling mechanism provided by GLIB to number each vertex. Recall that this vertex-labeling is simply a function from the vertices into some set; in our case, the set is $[|V| - n, |V|]$ at step n. We show below that this function is a bijection at each step of the algorithm, and we thus obtain the ordering σ by taking

the inverse of this labeling. At each step we will add one new vertex to this labeling and so we can use the domain of the vertex label to determine how many vertices we have numbered so far.

To store the lexicographic label we extend the graph library to support two different and independent vertex-labels. This second vertex-labeling will be used to order the vertices. For now it is enough to know that this second vertex-labeling will map to subsets of natural numbers. A graph with both vertex labels is referred to as a `VVGraph`.

We completely specify the LexBFS computation sequence by supplying an initial graph and a transition function. The initial graph is a `VVGraph` identical to the supplied graph with respect to vertices and edges, but with both vertex labels set to appropriate starting values. We completely ignore any edge-weights and edge-labels present in the original graph.

We use σ^{-1} to denote the first vertex-label (in order words, the vertex label is the inverse function of the ordering of the vertices). We will use L to denote the second vertex-label. When necessary, subscripts will be used to denote the labels in different steps of the algorithm. For example, $\sigma_j^{-1}(v) = k$ means that vertex v has number k in step j of the algorithm.

Definition 3.3.1 *Given a graph G, the function* `LexBFS:INIT(G)` *is a VVGraph whose vertices and edges are identical to G, but with $\sigma^{-1} = \emptyset$ and L such that for every $v \in V$, $L(v) = \emptyset$.*

```
definition let G be _Graph;
  func LexBFS:Init(G) -> natural-vlabeled finite-v2labeled
                         natsubset-v2labeled VVGraph equals
  G.set(VLabelSelector,{}).set(V2LabelSelector,
                               the_Vertices_of G-->{});
end;
```

The attribute *natural-vlabeled* means the first vertex-labeling (i.e., σ^{-1}) of the graph returned by `LexBFS:INIT` maps vertices to natural numbers, *finite-v2labeled* and *natsubset-v2labeled* ensure the second vertex-labeling (i.e. L) maps vertices to finite subsets of naturals.

The transition function will need to choose an unnumbered vertex to be numbered. This is done by the following function.

Definition 3.3.2 *Given a finite VVGraph G whose L maps to finite subsets of the naturals, if the domain σ^{-1} equals the vertices of G then* `LexBFS:PickUnnumbered(G)` *returns an arbitrary vertex, otherwise it returns a vertex, v, such that $L(v)$ is the largest with respect to the lexicographic ordering.*

```
definition let G be finite finite-v2labeled natsubset-v2labeled VVGraph;
  assume A: dom the_V2Label_of G = the_Vertices_of G;
  func LexBFS:PickUnnumbered(G) -> Vertex of G means
    it = choose the_Vertices_of G
         if dom the_VLabel_of G = the_Vertices_of G
  otherwise
    ex S being non empty finite Subset of bool NAT,
       B being non empty finite Subset of Bags NAT,
       F being Function
    st S = rng F &
```

```
      F = ((the_V2Label_of G) |
    (the_Vertices_of G \ dom the_VLabel_of G)) &
      (for x being finite Subset of NAT
         holds x in S implies ((x,1)-bag in B)) &
      (for x being set holds x in B implies
         ex y being finite Subset of NAT st y in S & x = (y,1)-bag) &
    it = choose (F " {support max(B,InvLexOrder NAT)});
end;
```

By lexicographic ordering above, we are refering to the lexicographic ordering on the subsets of natural numbers L. For example, let v and w be vertices such that $L(v) = \{12, 9, 8\}$ and $L(w) = \{12, 9, 7, 4\}$. The numbers in the these sets correspond to the numbers assigned to vertices adjacent to v or w by the algorithm on previous steps. To compare two subsets under the ordering, we compare the two largest elements, then the next two largest, etc, until there is a difference. In the above example, $L(v) > L(w)$, because $8 > 7$ and the two subsets are equal on the first two largest values. If $L(v) = \{12, 9, 8\}$ and $L(w) = \{12, 9, 8, 7, 4\}$, then $L(w) > L(v)$.

The MIZAR version of this definition is quite long and needs some explanation. The first case of this function handles the situation where all the vertices have been numbered, that is, when the algorithm is complete. Recall that a computation sequence is defined as an infinite sequence of graphs and so it needs to return a vertex even after the algorithm is already "finished". We use the MIZAR "choose" function to return an arbitrary vertex.

Everything after the "otherwise" in this definition is the heart of the algorithm. We convert each $L(v)$ for $v \in V$ into a bag of naturals (a bag in MIZAR is a multi-set). This is necessary because the lexicographic ordering we need is already defined on bags and not on subsets of naturals. Using this ordering we obtain the largest bag with respect to the ordering, and hence a vertex with the largest label.

Here is the entire process in more detail. We first restrict our attention to those vertices that are unnumbered by restricting L to those vertices not already in the domain of σ, this is the function F. The set S is the range of F and so contains finite subsets of natural numbers, the labels of the unnumbered vertices. Note this function is not one-to-one. We show that there is a set B containing exactly the bags of each element of S. We order the bags with the `InvLexOrder` and obtain the maximum bag with the `max` function. This bag is converted back into a finite subset of naturals via the `support` function. We take the pre-image of this set in F (that is, the vertices that map to this subset of naturals) and can now finally choose any vertex from this pre-image.

After a vertex has been chosen by the algorithm, the labels of its unnumbered neighbors are updated.

Definition 3.3.3 *Given a finite VVGraph G, a set v, and a natural k, we define* `LexBFS:LabelAdjacent(G, v, k)` *to be the function that returns a VVGraph with k added to the L of each of the unnumbered neighbors of v in G.*

```
definition let G be VVGraph, v be set, k be natural number;
  func LexBFS:LabelAdjacent(G, v, k) -> VVGraph equals
     G.set(V2LabelSelector,
           (the_V2Label_of G) .\/
           ((G.AdjacentSet({v}) \ dom the_VLabel_of G)-->{k}));
end;
```

Note: the .\/ function unions the images of elements in both functions.

After the neighbors of the chosen vertex have been updated we need to number the chosen vertex. Both of these operations are combined by the following function.

Definition 3.3.4 *Given an VGraph G, a vertex v, and a natural n, we define* `LexBFS:Update(G, v, n)` *to be the function returning a VVGraph with n added to the vertex-labeling of G and with n added to labels of the neighbors of v.*

```
definition let G be finite natural-vlabeled
                    finite-v2labeled natsubset-v2labeled VVGraph,
                    v be Vertex of G, n be natural number;
  set k = G.order()-'n;
  func LexBFS:Update(G, v, n) -> finite natural-vlabeled
                                 finite-v2labeled natsubset-v2labeled
                                 VVGraph equals
     LexBFS:LabelAdjacent(G.labelVertex(v, k), v, k);
end;
```

Notice we number v with k, which is defined with $k = G.order() -' n$. This is because n is the number of the current iteration, and we begin numbering from n and count down.

We are now in a position to define the transition function.

Definition 3.3.5 *Given a VVGraph G,* `LexBFS:Step(G)` *returns G itself if all vertices in G are numbered, otherwise it returns a graph equal to* `LexBFS:Update` *applied on G and* `LexBFS:PickUnnumbered(G)`.

```
definition let G be finite natural-vlabeled
                    finite-v2labeled natsubset-v2labeled VVGraph;
  func LexBFS:Step(G) -> finite natural-vlabeled
                         finite-v2labeled natsubset-v2labeled
                         VVGraph equals
    G if G.order() <= card (dom the_VLabel_of G) otherwise
                      LexBFS:Update(G,
                                    LexBFS:PickUnnumbered(G),
                                    card (dom the_VLabel_of G));
end;
```

With the initial graph and our transition function defined, we are now in a position to define the computation sequence.

Definition 3.3.6 *Given a finite graph G, we define* `LexBFS:CSeq(G)` *to be the sequence of VVGraphs where the first element is* `LexBFS:Init(G)`*, and every subsequent element in the sequence is obtained by applying* `LexBFS:Step` *to the element before it.*

```
definition let G be finite _Graph;
func LexBFS:CSeq(G) -> finite natural-vlabeled
                       finite-v2labeled natsubset-v2labeled
                       VVGraphSeq means
   it.0 = LexBFS:Init(G) &
   for n being natural number holds it.(n+1) = LexBFS:Step(it.n);
end;
```

This concludes the formal definitions required for the LexBFS algorithm.

3.3.4 Proving Correctness of LexBFS

We first show that LexBFS is a `vlabel-numbering` graph sequence. Let G be any finite graph.

Lemma 3.3.7 `LexBFS:CompSeq(G)` *is* halting.

Lemma 3.3.8 `LexBFS:CompSeq(G)` *is* iterative.

Both of the above lemmas follow immediately from Definition 3.3.6.

Lemma 3.3.9 `LexBFS:CompSeq(G)` *is* eventually-constant.

Proof. Immediately from Lemmas 3.3.7, 3.3.8 and Theorem 3.2.3. ∎

Lemma 3.3.10 *The* `Lifespan()` *of* `LexBFS:CompSeq(G)` *is equal to* `G.order()`.

Proof. Assume not, and let $n <$ `G.order()` be the assumed lifespan. By Definitions 3.3.6, 3.3.5, and 3.3.4 there will be some vertex w that will receive a number; i.e., there is some unnumbered vertex at this step of the algorithm. This means that the graphs at steps n and $n+1$ are not equal, and so step n cannot be the lifespan. Contradiction. ∎

We can now easily show:

Lemma 3.3.11 *For any graph G,* `LexBFS:CompSeq(G)` *is* vlabel-numbering.

Proof. Follows immediately from Definitions 3.3.1 to 3.3.6 and Lemmas 3.3.8, 3.3.9. ∎

Since `LexBFS:CompSeq(G)` is `vlabel-numbering`, we can use all the results from Section 3.2 to prove its correctness. However, in order to do that, we need to introduce an important property of vertex orders in order to prove that the ordering produced by LexBFS is a perfect vertex elimination scheme. This property is referred to in Golumbic[10] as property L3; we use the same name here.

Definition 3.3.12 (Property L3) *Let $G = (V, E)$ be a graph and σ an ordering of V. We say σ has Property L3 if for all vertices a, b, c such that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$ there is a vertex $d \in Adj(b) - Adj(a)$ with $\sigma^{-1}(c) < \sigma^{-1}(d)$.*

```
definition let G be natural-vlabeled VGraph;
  attr G is with_property_L3 means
   for a,b,c being Vertex of G st
       a in dom the_VLabel_of G &  b in dom the_VLabel_of G &
       c in dom the_VLabel_of G &
       (the_VLabel_of G).a < (the_VLabel_of G).b &
       (the_VLabel_of G).b < (the_VLabel_of G).c &
       a,c are_adjacent & not b,c are_adjacent
     ex d being Vertex of G st
       d in dom the_VLabel_of G &
       (the_VLabel_of G).c < (the_VLabel_of G).d &
       b,d are_adjacent & not a,d are_adjacent;
end;
```

In the remainder of this section we prove two main results. The first is that any ordering produced by LexBFS has property L3. The second is that any vertex ordering with property L3 on a chordal graph must be a perfect vertex elimination scheme.

To show that an order created by LexBFS has property L3 we need to prove a few lemmas. This first lemma says that we can place bounds on what values appear in a vertex's label at each step of the algorithm.

Lemma 3.3.13 *For a finite graph G, vertex v, and natural i, it holds that $L_i(v) \subseteq [|V| - i + 1, |V|]$.*

```
theorem tV202a:
for G being finite _Graph, i being natural number, v being set
  holds (the_V2Label_of (LexBFS:CompSeq(G).i)).v c=
        (Seg G.order()) \ Seg (G.order() -' i)
```

Note: function **Seg** n returns the set $[1, n]$ when n is a natural.

Proof. Let $x \in V$ be a vertex of G. We proceed by induction on k, the current step of the algorithm. For $k = 0$, $L_k(x) = \emptyset$ by Definition 3.2.6; so the base case holds trivially. Suppose for $i = k$ that $L_i(x) \subseteq [|V| - i + 1, |V|]$. Let v be the vertex chosen at step i. Since $[|V| - i + 1, |V|] \subset [|V| - (i + 1) + 1, |V|]$, we need only show that any value we add to $L_i(x)$ is in $[|V| - (i + 1) + 1, |V|]$. If x is numbered at step i or x is not adjacent to v, then $L_i(x) = L_{i+1}(x)$ by Definition 3.3.3. If x is unnumbered at step i and adjacent to v, then $L_{i+1}(x) = L_i(x) \cup \{\sigma^{-1}(v)\}$ again by Definition 3.3.3, and since $\sigma^{-1}(v) = |V| - i \in [|V| - (i + 1) + 1, |V|]$ the result is proved. ∎

Once a value is added to a vertex's label it is never removed. Consequently, a vertex's label in earlier steps of the algorithm will be a subset of its label in later steps.

Lemma 3.3.14 *For any finite graph G and vertex x, $L_i(x) \subseteq L_j(x)$, where $i \leq j$. That is, x's label is non-decreasing with respect to the lexicographic ordering as the algorithm proceeds.*

```
theorem V2Label3:
for G being finite _Graph, x being set,
    i,j being natural number st i <= j
 holds (the_V2Label_of (LexBFS:CompSeq(G).i)).x c=
       (the_V2Label_of (LexBFS:CompSeq(G).j)).x
```

Proof. The property is mentioned in Golumbic[10] where it is referred to as the L1 property. We proceed by induction on k. For $k = 0$, it is true that $L_i(x) \subseteq L_{i+k}(x)$. Suppose now that $L_i(x) \subseteq L_{i+k}(x)$. Let v be the vertex chosen at step i. If x is numbered at step i or not adjacent to v then $L_{i+k}(x) = L_{i+(k+1)}(x)$ by Definition 3.3.3 and so $L_i(x) \subseteq L_{i+(k+1)}(x)$. If x is unnumbered at step i and adjacent to v, then $L_{i+(k+1)}(x) = L_{i+k}(x) \cup \{\sigma^{-1}(v)\}$ and so $L_i(x) \subseteq L_{i+(k+1)}(x)$. Since $i \leq j$ there is some k such that $i + k = j$, and so it is true that $L_i(x) \subseteq L_j(x)$. ∎

There is only one opportunity to store a specific value in a label.

Lemma 3.3.15 *Let G be a finite graph, m, n, k be naturals such that $k < n \leq m$, and let x be a vertex of G. If $|V| - k \notin L_n(x)$ then $|V| - k \notin L_m(x)$.*

```
theorem V2Label4:
for G being finite _Graph,
    m,n,k being natural number st k < n & n <= m
for x being set
  st not G.order()-'k in (the_V2Label_of (LexBFS:CSeq(G).n)).x
    holds not G.order()-'k in (the_V2Label_of (LexBFS:CSeq(G).m)).x
```

Proof. We show by induction that if $|V| - k \notin L_{k+1}(x)$ then $|V| - k \notin L_{k+i}(x)$, let this result be $(*)$. The case when $i = 1$ is trivial. The inductive step follows since for any step $i + k > k$ we will either add $|V| - (i + k)$ to x's label or leave it alone; in any case we certainly do not add the value $|V| - k$. This proves $(*)$. Now, since $L_{k+1}(x) \subseteq L_n(x)$ by Lemma 3.3.14 and $|V| - k \notin L_n(x)$, it holds that $|V| - k \notin L_{k+1}(x)$. Hence by $(*)$ it follows that $|V| - k \notin L_m(x)$. ∎

Every value that appears in a vertex's label corresponds to some vertex adjacent to it that was given that value at an earlier step in the algorithm.

Lemma 3.3.16 *For a finite graph G, vertex x, and natural numbers m and n such that $n \in L_m(x)$, there is a vertex w adjacent to x such that w was chosen in step $|V| - n$.*

```
theorem V2Label7a1:
for G being finite _Graph, m,n being natural number
for x being Vertex of (LexBFS:CSeq(G).m)
 st n in (the_V2Label_of (LexBFS:CSeq(G).m)).x
  ex y being Vertex of (LexBFS:CSeq(G).m)
   st y = ((LexBFS:PickUnnumbered(LexBFS:CSeq(G).(G.order()-'n)))
    & not y in dom (the_VLabel_of (LexBFS:CSeq(G).(G.order()-'n)))
    & x in G.AdjacentSet({y}))
```

Proof. Let w be the vertex chosen in step $|V| - n$, and assume that w is not adjacent to x. Set $j = |V| - n$. By Lemma 3.3.13, $L_j(x) \subseteq [|V| - j + 1, |V|] = [|V| - |V| + n + 1, |V|] = [n + 1, |V|]$, and so $n \notin L_j(x)$. Since we assumed x is not adjacent to w, it follows that $n \notin L_{j+1}(x)$ by Definition 3.3.3. Since $j + 1 < m$ however, Lemma 3.3.15 implies that $n \notin L_m(x)$. Contradiction. ∎

We are finally in a position to prove that an order produced by LexBFS has property L3.

Theorem 3.3.17 *Let $G = (V, E)$ be a graph and σ_n the ordering produced by LexBFS up to the nth step. Then σ_n has property L3.*

```
theorem LexBFShasL3helper:
for G being finite _Graph, n being natural number
   holds LexBFS:CompSeq(G).n is with_property_L3
```

Proof. Let a, b, c be so that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$. Assume towards contradiction that for any vertex d adjacent to b such that $\sigma^{-1}(c) < \sigma^{-1}(d)$ it holds that d is also adjacent to a.

Let i be the step in which b was picked. It is then true that $\sigma^{-1}(c) \in L_i(a)$ and $\sigma^{-1}(c) \notin L_i(b)$ because c was chosen before both a and b and because c is adjacent to a and not adjacent to b. If $L_i(a)$ and $L_i(b)$ contain the same numbers greater than $\sigma^{-1}(c)$, then $L_i(a) > L_i(b)$, a contradiction (note this includes the case where $L_i(a)$ and $L_i(b)$ both contain no numbers larger than $\sigma^{-1}(c)$ as well). So there must exist some $z \in L_i(b)$ with $\sigma^{-1}(c) < z$ and $z \notin L_i(a)$ in order for $L_i(a) \leq L_i(b)$. But this implies the existence of a vertex y such that $\sigma^{-1}(y) = z$ that is adjacent to b and not to a. This is a contradiction to our initial assumption.

This proves the existence of at least one such vertex. From the nonempty set of these vertices choose the vertex x such that $\sigma^{-1}(x)$ is the largest. This vertex is guaranteed to meet the requirements of property L3. ∎

Corollary 3.3.18 *The ordering produced by LexBFS has property L3.*

```
theorem LexBFShasL3:
for G being finite _Graph
  holds (LexBFS:CompSeq(G)).Result() is with_property_L3
```

Before we can prove the final theorem of this section we need to prove a lemma on open chordless paths.

Lemma 3.3.19 *Let G be a chordal graph and P an open chordless path of G. If $x \in V$ is adjacent to the last vertex of P but not to the second last vertex of P, then P appended with x is also an open chordless path of G.*

```
for G being chordal _Graph, P being Path of G
   st P is open & P is chordless
for x,e being set st (not x in P.vertices() & e Joins P.last(),x,G &
                      not ex f being set st f Joins P.(len P-2),x,G)
  holds P.addEdge(e) is Path-like & P.addEdge(e) is open &
        P.addEdge(e) is chordless
```

Proof. Let $G = (V, E)$ be a chordal graph, $P = [p_1, \ldots, p_n]$ an open chordless path of G. Let $x \in V$ be so that x is adjacent to p_n but not to p_{n-1}. We will show that x is not adjacent to p_i for $i \in [1, n-1]$, thus showing that P appended with x is also an open chordless path.

We proceed by complete induction backwards from the end of P. The base case is complete, since for $k = 1$ it is true that x is not adjacent to all v_i with $i \in [n-k, n-1]$.

Suppose it is true for $j \in [1, k]$, with $1 < k < n$; that is x is not adjacent to v_i for $i \in [n-k, n-1]$, and assume that x is adjacent to $p_{n-(k+1)}$. We can then construct the cycle $C = [p_{n-(k+1)}, p_{n-k}, \ldots, p_n, x, p_{n-(k+1)}]$ with length at least four. Since G is a chordal graph, it follows that C must contain a chord e. By the inductive hypothesis we know that e cannot have x as an endpoint, and so we are forced to conclude that e has endpoints in the set $\{p_{n-k}, p_{n-k+1}, \ldots, p_n\}$. This is a contradiction since P is a chordless path. ∎

We can now prove:

Theorem 3.3.20 *Let G be a finite chordal graph and σ an ordering with property L3. Then σ is a perfect vertex elimination scheme.*

```
theorem :: Theorem 4.3, Golumbic p. 86
for G being finite chordal natural-vlabeled VGraph
  st G is with_property_L3 &
     dom the_VLabel_of G = the_Vertices_of G
for V being VertexScheme of G st V" = the_VLabel_of G
  holds V is perfect
```

Note: here `V"` means the inverse of `V`, which we can take because `V` is defined to be one-to-one.

Proof. Let $G = (V, E)$ be a finite chordal graph, let $n = |V|$, and let $\sigma = [v_1, \ldots, v_n]$ be an ordering with property L3. Assume towards contradiction that σ is not a perfect vertex elimination scheme for G.

Since σ is not a perfect vertex elimination scheme, there must be some vertex v_i that is not simplicial in the subgraph induced by $S = [v_i, v_{i+1}, \ldots, v_n]$. Let this vertex be v and let the subgraph induced by S be H. Since v is not simplicial in H, there must exist at least two neighbors of v which are non-adjacent to each other. Choose such a pair a and b so that b is chosen with $\sigma^{-1}(b)$ as large as possible.

Now assume we have an open chordless path $P = [p_1, p_2, \ldots, p_{k-1}, p_k]$ in G with the following properties: $\sigma^{-1}(p_1) > \sigma^{-1}(p_k) > \sigma^{-1}(p_2) > \sigma^{-1}(p_{k-1})$; for each $v_i \in P$ it holds that $\sigma^{-1}(v_i) < \sigma^{-1}(p_1)$; and for every vertex x of G such that x and p_2 are adjacent and x and p_{k-1} are not adjacent and $x \neq p_1$ holds $\sigma^{-1}(x) < \sigma^{-1}(p_1)$.

We will show the following: given a path P_1 with the above properties, we can construct a new path P_2 that is one vertex longer but which also has the same properties. From this we may conclude that we can build an arbitarily long path by iteration. This is clearly a contradiction since G is a finite graph. We will also show that we do indeed have such a path in G: the path composed of v, a, b from above along with another vertex that must exist from property L3. From all this we must conclude that the assumption that σ was not a perfect vertex elimination scheme was erroneous, and hence that σ is a perfect vertex elimination scheme.

Let P be a path that meets the requirements above. Then p_1 and p_k are not adjacent since P is an open and chordless path. Then there is a vertex, call it p_{k+1} that is adjacent to p_k but not to p_2 by property L3 on the vertices p_2, p_k, and p_1 as a, b, and c, respectively. Property L3 also ensures that $\sigma^{-1}(p_1) < \sigma^{-1}(p_{k+1})$ and that p_{k+1} has the largest $\sigma^{-1}(p_{k+1})$ out of all vertices in G that are adjacent to p_k and not to p_2. Now suppose p_{k+1} is adjacent to p_{k-1}; then by applying property L3 on the vertices p_{k-1}, p_2, and p_{k+1} we obtain a vertex d adjacent to p_2 and not adjacent to p_{k-1} such that $\sigma^{-1}(p_{k+1}) < \sigma^{-1}(d)$, but this contradicts the assumed maximality of $\sigma^{-1}(p_1)$ (the third property of P). Hence p_{k+1} is not adjacent to p_{k-1}, and since P is an open chordless path P appended with p_{k+1} is also an open chordless path by Lemma 3.3.19. Let Q be the reversal of P appended with p_{k+1}. Routine verification shows that Q has the three properties listed above, but that Q is a longer path than P.

We now show that such a path with these properties does exist in G. To do this, construct the path $P = [b, v, a]$. P is an open chordless path of G since a and b are not adjacent. Property L3 implies the existence of a vertex c such that $\sigma^{-1}(b) < \sigma^{-1}(c)$ with c adjacent to a and not to v. Then by Lemma 3.3.19 we know that P appended with c is an open chordless path. Let Q be the reversal of this path, so that $Q = [c, a, v, b]$. Then it is easy to verify that Q satisfies all three of the above properties.

This completes the proof since we have shown a contradiction. ∎

Combining Theorems 3.3.18 and 3.3.20 we obtain the following result.

Corollary 3.3.21 *The ordering produced by LexBFS on a chordal graph G is a perfect vertex elimination scheme for G.*

```
for G being finite chordal VVGraph holds
  (the_VLabel_of (LexBFS:CSeq(G)).Result())"
    is perfect VertexScheme of G
```

This ends our formal proof of correctness for LexBFS.

3.3.5 Discussion

1200 lines of miscellaneous graph theorems needed to be proven as background before we could begin formalizing LexBFS. An additional 800 lines were needed to introduce the second vertex label and a few additional facts on graph sequences. The vertex numbering facts took another 400 lines. To define and prove the correctness of the LexBFS algorithm, 2000 lines were needed. In all, this is around 4500 lines of MIZAR.

The proof of Theorem 3.3.20 took only 300 lines. This is simply because the proof as presented by us is easy to write in MIZAR. Initially however, we followed the proof presented in Golumbic[10] much more closely and it took over 1300 lines. The general approach of the two proofs is the same: a contradiction is created by generating an infinite one-to-one sequence in a finite graph. In the proof above, we create an open chordless path of arbitrary length having a few key properties and show we can extend it. Golumbic creates a sequence of arbitrary length having a few key properties; these properties allow him to use property L3 to extend the sequence and by building an open chordless path from this new sequence he shows that it also has those key properties. It turns out this sequence of vertices is not really needed. Without thinking about the matter too seriously we formalized Golumbic's proof and it turned out to be a nightmare to work with. The bulk of this proof revolved around extracting the path from the sequence (it was necessary to do this in three parts and then concatenate them together, which is a lot of tedious work), adding the new vertex, and then showing the path was chordless (to do this it is necessary to check every pair of indices the chord could lie on). Our proof completely avoids all this unpleasantness.

Again, the proof of a result that works nicely on paper can be a nightmare to do formally. It is usually preferable in cases like this to either amend the proof slightly so it is more suitable for formalization, or come up with an entirely new proof using the key ideas of the original.

3.4 Maximum Cardinality Search

3.4.1 The MCS algorithm

In this section we discuss another $O(|V| + |E|)$ algorithm for recognizing chordal graphs known as Maximum Cardinality Search (MCS).

The version of MCS that we will translate and verify in MIZAR is found in Tarjan[25] (see Algorithm 2 for a pseudo-code version). MCS takes any finite undirected graph $G = (V, E)$ as input and as output gives an ordering σ of the vertices. We need to prove σ is a perfect vertex elimination scheme only if G is chordal (the other direction is handled by Theorem 3.1.12).

MCS is similar to LexBFS. Vertices are numbered in the same way as in the LexBFS algorithm, the main difference is how the vertex to be numbered is chosen at each step. Instead of sorting the unnumbered vertices lexicographically by their labels and choosing a vertex with maximum label at each step, MCS stores a single non-negative integer value for each vertex: the number of its neighbours that have been numbered up to this point in the execution of the algorithm. Each vertex's label is initialized to zero, and all vertices are flagged as unnumbered. At each step of the algorithm an unnumbered vertex with the most numbered neighbours is chosen to be numbered and its neighbours' labels are incremented by one.

Algorithm 2 Maximum Cardinality Search

Require: Finite undirected graph $G = (V, E)$
Ensure: Ordering σ of V
1: Assign the label 0 to each vertex
2: **for** $i = |V|$ downto 1 **do**
3: $v \leftarrow$ unnumbered vertex with largest label
4: $\sigma(i) = v$
5: **for all** unnumbered vertices $w \in Adj(v)$ **do**
6: label(w) = label(w) + 1
7: **end for**
8: **end for**
9: **return** σ

MCS is straightforward to implement. To obtain an $O(|V|^2)$ runtime, we can keep the vertex labels in an array and simply run through it to find the vertex with the largest label at each step.

Again, as in the case for LexBFS, these implementation details are of no concern to us since we currently do not have the machinery to deal with their time complexity issues in MIZAR. We simply show that the algorithm terminates after exactly $|V|$ steps, that is, after each vertex as been numbered.

We now discuss our formalization of MCS in MIZAR.

3.4.2 Formal Definitions of MCS

As with LexBFS, we will store all the information required during the algorithm in vertex labels. We define MCS as a vertex numbering graph sequence as in Section 3.2. After the algorithm is complete, the vertex label will map each vertex to its position in the ordering. The number of neighbors that have been numbered will be stored in the second vertex-label that we used in LexBFS for the lexicographic label; here, however, we need only store a natural number.

We completely specify the MCS computation sequence by supplying an initial graph and a transition function. The initial graph is a `VVGraph` identical to the supplied graph with respect to vertices and edges, but with both vertex labels erased

and set to appropriate starting values. We completely ignore the edge-weighting and edge-labeling present in the original graph.

Definition 3.4.1 *Given a graph G, the function* `MCS:Init(G)` *returns a* `VVGraph` *in which $\sigma^{-1} = \emptyset$ and for $v \in V$, $L(v) = 0$.*

```
definition let G be finite _Graph;
  func MCS:Init(G) -> finite natural-vlabeled natural-v2labeled
                      VVGraph equals
    G.set(VLabelSelector, {}).set(V2LabelSelector,
                                  the_Vertices_of G --> 0);
end;
```

The `natural-vlabled` and `natural-v2labeled` properties say the returned graph's vertex labels each map to the naturals. The first vertex label (σ^{-1}) will hold the position of each vertex in the vertex order, the second label (L) will hold the current number of neighbors that have been numbered so far.

The transition function will need to pick an unnumbered vertex to be numbered. The following function returns such a vertex.

Definition 3.4.2 *Given a finite* `VVGraph` *G whose second vertex labeling maps to the naturals, if the domain of the second labeling equals the vertices of G then* `MCS:PickUnnumbered(G)` *returns an arbitrary vertex, otherwise it returns a vertex with the largest label.*

```
definition let G be finite natural-v2labeled VVGraph;
  assume A: dom the_V2Label_of G = the_Vertices_of G;

  func MCS:PickUnnumbered(G) -> Vertex of G means
    it = choose the_Vertices_of G
      if dom the_VLabel_of G = the_Vertices_of G
    otherwise
      ex S being finite non empty natural-membered set,
        F being Function st S = rng F
          & F = (the_V2Label_of G) |
                (the_Vertices_of G \ dom the_VLabel_of G)
          & it = choose (F " {max S});
end;
```

If all vertices have been numbered (the domain of the vertex label equals the vertices of G) then some arbitrary is returned. If there are some unnumbered vertices then we restrict the second vertex labeling to the unnumbered vertices and call this restriction F. The range of F is a subset of the naturals, but the `max` function requires a non empty set of naturals. So it is necessary to instantiate the set S with the non empty attribute. MIZAR does not know automatically that the range of F is non empty, but the justification follows in a single step (since the domain of F is non empty the range must be as well). An arbitrary vertex in the pre-image of the maximum value of S is returned.

After a vertex has been chosen each neighbor needs to have its label incremented by one. The following function returns a new graph with the neighbors updated appropriately.

Definition 3.4.3 *Given a finite graph G whose second vertex labeling maps to the naturals and a set v,* `MCS:LabelAdjacent(G, v)` *returns the graph in which the unnumbered neighbors of v have their labels incremented by one.*

```
definition let G be finite natural-v2labeled VVGraph, v be set;
  func MCS:LabelAdjacent(G, v) -> finite natural-v2labeled
                                  VVGraph equals
    G.set(V2LabelSelector,
          (the_V2Label_of G).incSubset(G.AdjacentSet({v})
                                       \ dom (the_VLabel_of G),1));
end;
```

This definition requires the `incSubset` function, which is defined as follows:

Definition 3.4.4 *Given a natural yielding function F, a set S, and a natural k, the function* `incSubset(S,k)` *returns a function G such that for* $x \notin S$ $G.x = F.x$ *and for* $x \in S$ $G.x = F.x + k$.

```
definition let F be natural-yielding Function,
               S be set, k be natural number;

   func F.incSubset(S,k) -> natural-yielding Function
                            means   :dIncSubset:
     dom it = dom F &
     for y being set holds (y in S & y in dom F implies it.y = F.y + k)
                         & (not y in S implies it.y = F.y);
end;
```

Note that in MIZAR, if x is not in the domain of F then $F.x = 0$ by definition.

`MCS:LabelAdjacent(G,v)` uses definition 3.4.4 to increment the set of neighbors restricted to those that are unnumbered.

`MCS:LabelAdjacent` updates the neighbors, but the chosen vertex still needs to be numbered. The following function numbers the given vertex with the value `G.order() -' n`. Here n is assumed to be the current step of the algorithm and so the vertices are numbered in decreasing order.

Definition 3.4.5 *Given a finite graph G with natural yielding vertex labels, a vertex v, and a natural n,* `MCS:Update(G, v, n)` *returns a graph with v numbered with* `G.order()-'n` *and the labels of the neighbors of v incremented.*

```
definition
  let G be finite natural-vlabeled natural-v2labeled VVGraph,
      v be Vertex of G, n be natural number;
    func MCS:Update(G, v, n) -> finite natural-vlabeled
                                natural-v2labeled VVGraph equals

      MCS:LabelAdjacent(G.labelVertex(v, G.order()-'n), v);

end;
```

The next function is the actual transition function of MCS.

Definition 3.4.6 *Given a finite graph G with natural yielding vertex labels and a natural n,* `MCS:Step(G)` *returns G if all vertices have been numbered, otherwise it returns* `MCS:Update(G, MCS:PickUnnumbered(G), n)`.

```
definition let G be finite natural-vlabeled
                    natural-v2labeled VVGraph,
                n be natural number;
  func MCS:Step(G) -> finite natural-vlabeled natural-v2labeled
                      VVGraph equals
    G if G.order() <= n otherwise
    MCS:Update(G, MCS:PickUnnumbered(G), n);
end;
```

We can now define the MCS computation sequence.

Definition 3.4.7 *Given a finite graph G,* `MCS:CSeq(G)` *returns the graph sequence with the first element equal to* `MCS:Init(G)` *and with each subsequent element equal to* `MCS:Step` *applied on the element before it.*

```
definition let G be finite _Graph;
  func MCS:CSeq(G) -> finite natural-vlabeled
                      natural-v2labeled VVGraphSeq  means
    it.0 = MCS:Init(G) &
    for n being natural number
        holds it.(n+1) = MCS:Step(it.n);
end;
```

This concludes the formal definitions required for the MCS algorithm.

3.4.3 Proving Correctness of MCS

This section will follow the proof layout of Section 3.3.4 closely. We first show MCS is a `vlabel-numbering` graph sequence, introduce a property on vertex orders, show that an order produced by MCS must have this property, and finally show that any order with this property on a chordal graph is a perfect vertex elimination scheme.

The notational convention of Section 3.3.4 will be used here. The ordering on the numbered vertices of the nth graph in the computation sequence will be denoted with σ_n, the subscript will be dropped if the particular step of the sequence is not important. The mth vertex of an order is written as $\sigma(m)$, and the number given to a vertex x is then $\sigma^{-1}(x)$ (since the ordering is one-to-one). The number of adjacent numbered vertices for a vertex x is written as $L(x)$, and the step of the algorithm is specified as a subscript (e.g. $L_n(x)$ for the label of x in the nth step of the sequence).

Lemma 3.4.8 *For any graph G,* `MCS:CSeq(G)` *is* `vlabel-numbering`.

Proof. The proof is very similar to that of Lemma 3.3.11. ∎

For the proof of correctness of MCS we follow the proof by Tarjan found in Tarjan[25]. Tarjan defines the following property on vertex orders (he names it Property P but we have given it the name Property T).

Definition 3.4.9 (Property T) *Let $G = (V, E)$ be a graph and σ an ordering of V. We say σ has Property T if for all vertices a, b, c such that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$ there is a vertex $d \in Adj(b) - Adj(a)$ with $\sigma^{-1}(b) < \sigma^{-1}(d)$.*

```
definition let G be natural-vlabeled VGraph;
  attr G is with_property_T means  :dPropertyT:
for a,b,c being Vertex of G st a in dom the_VLabel_of G &
    b in dom the_VLabel_of G & c in dom the_VLabel_of G &
    (the_VLabel_of G).a < (the_VLabel_of G).b &
    (the_VLabel_of G).b < (the_VLabel_of G).c &
    a,c are_adjacent & not b,c are_adjacent
  ex d being Vertex of G st d in dom the_VLabel_of G &
     (the_VLabel_of G).b < (the_VLabel_of G).d &
     b,d are_adjacent & not a,d are_adjacent;
end;
```

Notice this is almost exactly the same as Property L3 (Definition 3.3.12) except that the clause "$\sigma^{-1}(b) < \sigma^{-1}(d)$" has replaced "$\sigma^{-1}(c) < \sigma^{-1}(d)$". This means that Property T is a generalization of Property L3, and hence:

Corollary 3.4.10 *For G being a finite graph, the order produced by* `LexBFS:CSeq(G)` *has property T.*

Proof. Immediate from the above definition and Corollary 3.3.18. ∎

Once we have proven the correctness of MCS, we will also have proven the correctness of LexBFS, for the second time. This does not mean however, that MCS is a generalization of LexBFS; there are orders produced by LexBFS that are not produced by MCS and vice-versa.

The small difference between Property T and Property L3 necessitates a new proof since the proof for LexBFS requires the assumption that vertex d (whose existence is guaranteed by T and L3) be chosen before c (not just before b). This is actually a good thing. We can stress test the robustness of our formalization and fill in any gaps in facts that we have already developed.

Lemma 3.4.11 *For an unnumbered vertex x in step n of the MCS algorithm, $L_n(x)$ is the number of numbered vertices adjacent to x.*

```
theorem MCSuV2Label2:
for G being finite _Graph, n being natural number, x being set
 st not x in (dom the_VLabel_of (MCS:CSeq(G).n))
  holds (the_V2Label_of (MCS:CSeq(G).n)).x =
        card (G.AdjacentSet({x}) /\
             (dom the_VLabel_of (MCS:CSeq(G).n)))
```

Proof. This lemma is obvious to human readers, but takes a modest amount of work to do formally. We proceed by induction on k, the step of the algorithm. The result holds for $k = 0$ by Definitions 3.4.7 and 3.4.1. Suppose it is true for some $k > 0$. Let w be the vertex chosen by the algorithm in step k. Let x be any unnumbered vertex in step $k+1$ (and so $x \neq w$). Set N_k and N_{k+1} to the sets of numbered vertices in steps k and $k+1$ respectively. Then we know that $N_{k+1} = N_k \cup \{w\}$ by Defintion 3.4.5. By the induction hypothesis we know $|N_k \cap Adj(x)| = L_k(x)$. Suppose now

that x is adjacent to w. Then $L_{k+1}(x) = L_k(x) + 1$ by Definition 3.4.3. Then,

$$\begin{aligned} |N_{k+1} \cap Adj(x)| &= |(N_k \cup \{w\}) \cap Adj(x)| \\ &= |(N_k \cap Adj(x)) \cup (\{w\} \cap Adj(x))| \\ &= |(N_k \cap Adj(x)) \cup \{w\}| \\ &= |N_k \cap Adj(x)| + 1 \\ &= L_k(x) + 1 \\ &= L_{k+1}(x). \end{aligned}$$

Suppose instead that x is not adjacent to w. Then $L_{k+1}(x) = L_{k+1}(x)$ by Definition 3.4.3. And so

$$\begin{aligned} |N_{k+1} \cap Adj(x)| &= |(N_k \cup \{w\}) \cap Adj(x)| \\ &= |(N_k \cap Adj(x)) \cup (\{w\} \cap Adj(x))| \\ &= |(N_k \cap Adj(x)) \cup \emptyset| \\ &= |N_k \cap Adj(x)| \\ &= L_k(x) \\ &= L_{k+1}(x). \end{aligned}$$

This proves the result. ∎

We can now show:

Theorem 3.4.12 *Let $G = (V, E)$ be a graph and σ_n the ordering produced by MCS at the nth step. Then σ_n has property T.*

```
theorem MCShasThelper:
for G being finite _Graph, n being natural number
   holds MCS:CSeq(G).n is with_property_T
```

Proof. Let σ be an ordering produced by MCS and let a, b, c be such that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$. Let X be the set of vertices chosen before b, that is, those vertices $x \in X$ such that $\sigma^{-1}(b) < \sigma^{-1}(x)$. Divide X into four disjoint sets X_1, X_2, X_3 and X_4, where X_1 contains vertices adjacent to b and not to a, X_2 contains vertices adjacent to a and not to b, X_3 contains vertices adjacent to both a and b, and X_4 the remaining vertices. We know $|X_2| \geq 1$ since $c \in X_2$.

Since $\sigma^{-1}(a) < \sigma^{-1}(b)$, b was chosen before a and so $|X_1| + |X_3| \geq |X_2| + |X_3|$. This implies that $|X_1| \geq 1$ and so the result is proved. ∎

We prove a simple lemma on vertex schemes before the final result of this section.

Lemma 3.4.13 *Let σ be an ordering of G. Then σ is a perfect vertex elimination scheme of G if and only if for any a, b, c such that $ab \in E$, $ac \in E$, $\sigma^{-1}(a) < \sigma^{-1}(b)$ and $\sigma^{-1}(a) < \sigma^{-1}(c)$, it holds that $bc \in E$.*

```
theorem :: CHORD:109
for G being finite _Graph, V being VertexScheme of G holds
  V is perfect iff
  for a,b,c being Vertex of G
```

```
    st b<>c & a,b are_adjacent & a,c are_adjacent
  for va,vb,vc being natural number
    st va in dom V & vb in dom V & vc in dom V &
       V.va = a & V.vb = b & V.vc = c & va < vb & va < vc
    holds b,c are_adjacent;
```

Proof. For convenience, let G_x refer to the subgraph of G induced by x and all vertices following x in σ. If σ is a perfect elimination scheme, then the neighborhood of a in G_a is complete, and so $bc \in E$. Going the other way, if every such triplet has the above property, then for any vertex x it is true that x's neighborhood in G_x is complete, and hence σ is a perfect vertex elimination scheme of G. ∎

We can now prove:

Theorem 3.4.14 *Let $G = (V, E)$ be a chordal graph and let σ be an ordering of G. If σ has property T then σ is a perfect vertex elimination scheme.*

```
for G being finite chordal natural-vlabeled VGraph
  st G is with_property_T &
     dom the_VLabel_of G = the_Vertices_of G
   for V being VertexScheme of G
     st V" = the_VLabel_of G
holds V is perfect
```

Proof. This proof is from Tarjan[25]. Assume σ has property T. Consider all chordless paths of G with at least three vertices that have a certain property Q. From these paths, let $P = (v_0, v_1, v_2, \ldots v_k)$ be a path with maximum $\sigma^{-1}(v_k)$. We define property Q as follows: for some i in the interval $[1, k-1]$, it is true that

$$\sigma^{-1}(v_0) > \sigma^{-1}(v_k) > \sigma^{-1}(v_1) > \sigma^{-1}(v_2) \cdots > \sigma^{-1}(v_i)$$

and

$$\sigma^{-1}(v_i) < \sigma^{-1}(v_{i+1}) < \cdots < \sigma^{-1}(v_k)$$

hold.

Apply property T to vertices v_1, v_k, and v_0 to obtain a vertex x such that x is adjacent to v_k and $\sigma^{-1}(v_k) < \sigma^{-1}(x)$. Since there is at least one vertex adjacent to x (ie, v_k), let $j > 1$ be the minimum such that v_j is a vertex on P that is adjacent to x (if $j = 0$ then we would have a chordless cycle of length more than three, ie, $[x, v_0, \ldots, vk, x]$; $j \neq 1$ from property T). Now, if $\sigma^{-1}(v_0) > \sigma^{-1}(x)$, then the path $v_0, v_1, \ldots, v_j, x$ has property Q. If $\sigma^{-1}(x) > \sigma^{-1}(v_0)$, then the path $x, v_j, v_{j-1}, \ldots, v_0$ has property Q. In both cases we have a contradiction (the last vertex would be larger than v_k with respect to the ordering σ). From this we conclude that no path has property Q.

Now consider any triplet of vertices a,b,c with $ab \in E$, $ac \in E$, and so that σ orders a before both b and c. If $bc \notin E$, then one of bac or cab must have property Q. But since we have shown there cannot be any paths with property Q, it follows that no such triplets exist. Hence σ is a perfect vertex elimination scheme of G by Lemma 3.4.13. ∎

Combining Theorems 3.4.12 and 3.4.14 we obtain the following result.

Corollary 3.4.15 *The ordering produced by MCS on a chordal graph G is a perfect vertex elimination scheme for G.*

```
for G being finite chordal VVGraph holds
  (the_VLabel_of (MCS:CSeq(G)).Result())"
    is perfect VertexScheme of G
```

This ends our formal proof of correctness for MCS.

3.5 Discussion

To define and prove the correctness of MCS required only around 1600 lines of MIZAR. This is mainly because the definition of MCS is much easier to deal with than that of LexBFS. The fact that an ordering produced by MCS has property T was also much easier to prove than that an ordering produced by LexBFS has property L3 (around 90 lines compared to around 250 lines of code). The proof by Tarjan (Theorem 3.4.14) is also quite nice, and led us to go back and re-do the proof of Theorem 3.3.20 in a manner more suitable for MIZAR.

Chapter 4

Conclusion and Future Work

Extending the current library of graph functions and definitions was the main goal of this paper. We formalized the notion of a chordal graph and proved a characterization for this graph class. We also defined and proved the correctness for the Lexicograph-Breadth First Search and Maximum Cardinality Search algorithms, which are used to recognize chordal graphs. The relative ease with which these notions were formalized shows that the graph library is robust and suitable for dealing with graph algorithms.

There is much future work to be done. Proving that a chordal graph is the intersection graph of a family of subtrees of a tree is the next interesting result in the area of chordal graphs. Note that this would require the notion of intersection graphs to be defined in MIZAR. Some other areas in graph theory that are feasible for formalization include perfect graphs and graph isomorphism.

Bibliography

[1] ACL2. `http://pvs.csl.sri.com/`.

[2] The COQ Proof Assistant. `http://coq.inria.fr/`.

[3] The HOL Theorem Proving System. `http://hol.sourceforge.net/`.

[4] The Mizar Project. `http://mizar.org/`.

[5] PVS Specification and Verification System. `http://pvs.csl.sri.com/`.

[6] Jean-Raymond Abrial, Dominique Cansell, and Dominique Mry. Formal derivation of spanning trees algorithms.

[7] R.W Butler and J.A. Sjogren. A PVS Graph Theory Library. *59 Mathematical and Computer Sciences (General)*, February 1998.

[8] Jing-Chao Chen. Dijkstra's Shortest Path Algorithm. *Formalized Mathematics*, 15, 2003.

[9] Ranan Fraer. Formal development in b of a minimum spanning tree algorithm. Proceedings of the First B Conference, 1996.

[10] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 111 Fifth Avenue, New York, New York, 1980.

[11] Krzysztof Hryniewiecki. Graphs. *Formalized Mathematics*, 2, 1990.

[12] Gilbert Lee. Verification of Graph Algorithms in Mizar. Master's thesis, Unversity of Alberta, 2004.

[13] Gilbert Lee. Proof of Dijkstra's Shortest Path Algorithm and Prim's Minnimum Spanning Tree Algorithm. *Formalized Mathematics*, 16, 2005.

[14] Gilbert Lee. Proof of Ford/Fulkerson's Maximum Network Flow Algorithm. *Formalized Mathematics*, 16, 2005.

[15] Gilbert Lee. Trees: Connected, Acyclic Graphs. *Formalized Mathematics*, 16, 2005.

[16] Gilbert Lee. Weighted and Labeled Graphs. *Formalized Mathematics*, 16, 2005.

[17] Gilbert Lee and Piotr Rudnicki. Alternative Graph Structures. *Formalized Mathematics*, 16, 2005.

[18] Gilbert Lee and Piotr Rudnicki. Walks in a Graph. *Formalized Mathematics*, 16, 2005.

[19] J Strother Moore and Qiang Zhang. Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2. `http://www.cs.utexas.edu/users/moore/publications/dijkstra/index.html`.

[20] Yatsuka Nakamura and Jing-Chao Chen. The Underlying Principle of Dijkstra's Shortest Path Algorithm. *Formalized Mathematics*, 15, 2003.

[21] Yatsuka Nakamura and Piotr Rudnicki. Vertex Sequence Induced by Chains. *Formalized Mathematics*, 7, 1995.

[22] Yatsuka Nakamura and Piotr Rudnicki. Euler Circuits and Paths. *Formalized Mathematics*, 9, 1997.

[23] Yatsuka Nakamura and Piotr Rudnicki. Oriented Chains. *Formalized Mathematics*, 10, 1998.

[24] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame Graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 21–35. Springer, 2006.

[25] Robert E. Tarjan and Mihalis Yannakakis. Simplie linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13(3):566–569, August 1984.

[26] Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. Formalization of Graph Search Algorithms and its Applictions.

Appendix A

Introduction

Appendices B and C contain two different versions of the first three sections from the chapter on chordal graphs found in Golumbic[10]. The first version, Appendix B, is the original version found in the book; the second version, Appendix C, is our revised version. They have been interleaved so that the differences are more readily apparent.

The motivation for this is to demonstrate the incompletness and subtle inaccuracies of typical non-formalized proofs, and to show how the chapter would look once these considerations had been taken into effect.

Problems with the original version are marked with footnotes. Changebars mark the location of changes in the revised version.

We have also added a few lemmas and theorems dealing with MCS to the revised version that the original version noted only in passing.

Appendix B

Triangulated Graphs: Golumbic

1. Introduction

One of the first classes of graphs to be recognized as being perfect was the class of triangulated graphs. Hajnal and Suranyi [1958] showed that triangulated graphs satisfy the *perfect property* P_2 (α-perfection), and Berge [1960] proved that they satisfy P_1 (χ-perfection). These two results, in large measure, inspired the conjecture that P_1 and P_2 were equivalent, a statement that we now know to be true (Theorem 3.3). Thus, the study of triangulated graphs can well be thought of as the beginning of the theory of perfect graphs.

We briefly looked at the triangulated graph property in the sneak preview Section 1.3. For completeness' sake, we shall repeat the definition here and mention a few basic properties.

An undirected graph G is called *triangulated* if every cycle of length strictly greater than 3 possesses a chord, that is, an edge joining two nonconsecutive vertices of the cycle. Equivalently, G does not contain an induced subgraph isomorphic to C_n for $n > 3$. Being triangulated is a hereditary property inherited by all the induced subgraphs of G. You may recall from Section 1.3 that the interval graphs constitute a special type of triangulated graph. Thus we have our first example of triangulated graphs.

In the literature, triangulated graphs have also been called *chordal*, *rigid-circuit*, *monotone transitive*, and *perfect elimination* graphs.

2. Characterizing Triangulated Graphs

A vertex x of G is called *simplicial* if its adjacency set $Adj(x)$ induces a complete subgraph of G, i.e., $Adj(x)$ is a clique (not necessarily maximal). Dirac [1961], and later Lekkerkerker and Boland [1962], proved that a triangulated graph always has a simplicial vertex (in fact at least two of them), and using this fact Fulkerson and Gross [1965] suggested an iterative procedure to recognize triangulated graphs based on this and the hereditary property. Namely, *repeatedly locate a simplicial vertex and eliminate it from the graph, until either no vertices remain and the graph is triangulated or at some stage no simplicial vertex exists and the graph is not triangulated.* The correctness of this procedure is proved in Theorem 4.1. Let us state things more algebraically.

Appendix C

Triangulated Graphs: Revised

1. Introduction

One of the first classes of graphs to be recognized as being perfect was the class of triangulated graphs. Hajnal and Suranyi [1958] showed that triangulated graphs satisfy the *perfect property* P_2 (α-perfection), and Berge [1960] proved that they satisfy P_1 (χ-perfection). These two results, in large measure, inspired the conjecture that P_1 and P_2 were equivalent, a statement that we now know to be true (Theorem 3.3). Thus, the study of triangulated graphs can well be thought of as the beginning of the theory of perfect graphs.

We briefly looked at the triangulated graph property in the sneak preview Section 1.3. For completeness' sake, we shall repeat the definition here and mention a few basic properties.

An undirected graph G is called *triangulated* if every cycle of length strictly greater than 3 possesses a chord, that is, an edge joining two nonconsecutive vertices of the cycle. Equivalently, G does not contain an induced subgraph isomorphic to C_n for $n > 3$. Being triangulated is a hereditary property inherited by all the induced subgraphs of G. You may recall from Section 1.3 that the interval graphs constitute a special type of triangulated graph. Thus we have our first example of triangulated graphs.

In the literature, triangulated graphs have also been called *chordal*, *rigid-circuit*, *monotone transitive*, and *perfect elimination* graphs.

2. Characterizing Triangulated Graphs

A vertex x of G is called *simplicial* if its adjacency set $Adj(x)$ induces a complete subgraph of G, i.e., $Adj(x)$ is a clique (not necessarily maximal). Dirac [1961], and later Lekkerkerker and Boland [1962], proved that a triangulated graph always has a simplicial vertex (in fact at least two of them), and using this fact Fulkerson and Gross [1965] suggested an iterative procedure to recognize triangulated graphs based on this and the hereditary property. Namely, *repeatedly locate a simplicial vertex and eliminate it from the graph, until either no vertices remain and the graph is triangulated or at some stage no simplicial vertex exists and the graph is not triangulated.* The correctness of this procedure is proved in Theorem 4.1. Let us state things more algebraically.

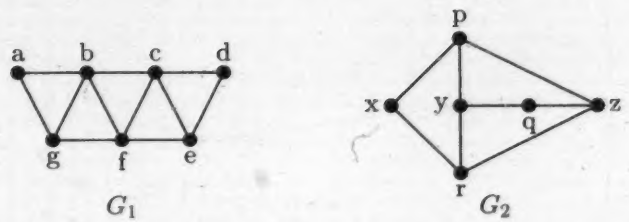

Figure A.1: Two graphs, one triangulated and one not triangulated.

Let $G = (V, E)$ be an undirected graph and let $\sigma = [v_1, v_2, \ldots, v_n]$ be an ordering of the vertices. We say that σ is a *perfect vertex elimination scheme* (or *perfect scheme*) if each v_i is a simplicial vertex of the induced subgraph $G_{\{v_i,\ldots,v_n\}}$. In other words, each set

$$X_i = \{v_j \in Adj(v_i) | j > i\}$$

is complete. For example, the graph G_1 in Figure 4.1 has a perfect vertex elimination scheme $\sigma = [a, g, b, f, c, e, d]$. It is not unique; in fact G_1 has 96 different perfect elimination schemes. In contrast to this, the graph G_2 has no simplicial vertex, so we cannot even start constructing a perfect scheme — it has none.

A subset $S \subseteq V$ is a *vertex separator* for nonadjacent vertices a and b (or an *a-b separator*) if the removal of S from the graph separates a and b into distinct connected components. If no proper subset of S is an a-b separator, then S is a *minimal vertex separator* for a and b.[1] Consider again the graphs of Figure 4.1. In G_2, the set $\{y, z\}$ is a minimal vertex separator for p and q, whereas $\{x, y, z\}$ is a minimal vertex separator for p and r. (How is it possible that both are *minimal* vertex separators, yet one is contained in the other?) In G_1, every minimal vertex separator has cardinality 2. This is an unusual phenomenon. However, notice also that the two vertices of such a separator of G_1 are adjacent, in every case. This latter phenomenon actually occurs for all triangulated graphs, as you will see in Theorem 4.1.

We now give two characterizations of triangulated graphs, one algorithmic (Fulkerson and Gross [1965]) and the other graph theoretic (Dirac [1961]).

Theorem 4.1 Let G be an undirected graph. The following statements are equivalent:

(i) G is triangulated.

(ii) G has a perfect vertex elimination scheme. Moreover, *any* simplicial vertex can start a perfect scheme.

(iii) Every minimal vertex separator induces a complete subgraph of G.

Proof. (iii) $\Rightarrow$ (i) Let $[a, x, b, y_1, y_2, \ldots, y_k, a]$ $(k \geq 1)$ be a simple cycle of $G = (V, E)$. Any minimal a-b separator must contain vertices x and y_i for some i, so $xy_i \in E$, which is a chord of the cycle.

[1] See our edition for some additional properties of vertex separators.

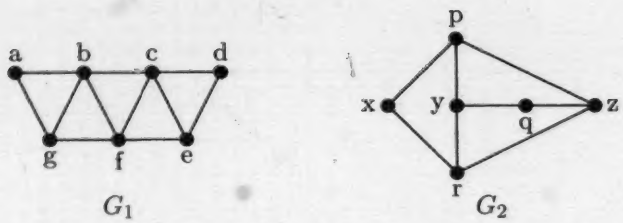

Figure B.2: Two graphs, one triangulated and one not triangulated.

Let $G = (V, E)$ be an undirected graph and let $\sigma = [v_1, v_2, \ldots, v_n]$ be an ordering of the vertices. We say that σ is a *perfect vertex elimination scheme* (or *perfect scheme*) if each v_i is a simplicial vertex of the induced subgraph $G_{\{v_i,\ldots,v_n\}}$. In other words, each set

$$X_i = \{v_j \in Adj(v_i) | j > i\}$$

is complete. For example, the graph G_1 in Figure 4.1 has a perfect vertex elimination scheme $\sigma = [a, g, b, f, c, e, d]$. It is not unique; in fact G_1 has 96 different perfect elimination schemes. In contrast to this, the graph G_2 has no simplicial vertex, so we cannot even start constructing a perfect scheme — it has none.

A subset $S \subseteq V$ is a *vertex separator* for nonadjacent vertices a and b (or an *a-b separator*) if the removal of S from the graph separates a and b into distinct connected components. Note that if a and b are already separated (in different components of a disconnected graph) then the empty set is an a-b separator. If no proper subset of S is an a-b separator, then S is a *minimal vertex separator* for a and b. For any two nonadjacent vertices there is always at least one minimal vertex separator. Consider again the graphs of Figure 4.1. In G_2, the set $\{x, y, z\}$ is a minimal vertex separator for p and q, whereas $\{x, y, z\}$ is a minimal vertex separator for p and r. (How is it possible that both are *minimal* vertex separators, yet one is contained in the other?) In G_1, every minimal vertex separator has cardinality 2. This is an unusual phenomenon. However, notice also that the two vertices of such a separator of G_1 are adjacent, in every case. This latter phenomenon actually occurs for all triangulated graphs, as you will see in Theorem 4.1.

We now give two characterizations of triangulated graphs, one algorithmic (Fulkerson and Gross [1965]) and the other graph theoretic (Dirac [1961]).

Theorem 4.1 Let G be an undirected graph. The following statements are equivalent:

(i) G is triangulated.

(ii) G has a perfect vertex elimination scheme. Moreover, *any* simplicial vertex can start a perfect scheme.

(iii) Every minimal vertex separator induces a complete subgraph of G.

Proof. (iii) $\Rightarrow$ (i) Let $[a, x, b, y_1, y_2, \ldots, y_k, a]$ $(k \geq 1)$ be a simple cycle of $G = (V, E)$. Any minimal a-b separator must contain vertices x and y_i for some i, so $xy_i \in E$, which is a chord of the cycle.

(i) $\Rightarrow$ (iii) Suppose S is a minimal $a-b$ separator with G_A and G_B being the connected components of G_{V-S} containing a and b, respectively. Since S is minimal, each $x \in S$ is adjacent to some vertex in A and some vertex in B. Therefore, for any pair $x, y \in S$ there exist paths $[x, a_1, \ldots, a_r, y]$ and $[y, b_1, \ldots, b_t, x]$, where each $a_i \in A$ and $b_i \in B$, such that these paths are chosen to be of smallest possible length. It follows that $[x, a_1, \ldots, a_r, y, b_1, \ldots, b_t, x]$ is a simple cycle whose length is at least 4, implying that it must have a chord. But $a_i b_j \notin E$ by the definition of vertex separator, and $a_i a_j \notin E$ and $b_i b_j \notin E$ by the minimality of r and t.[2] Thus, the only possible chord is $xy \in E$. $\square$

Remark. It also follows that $r = t = 1$, implying that for all $x, y \in S$ there exist vertices in A and B which are adjacent to both x and y. A stronger result is given in Exercise 12.[3]

Before continuing with the remaining implications, we pause for a message from our lemma department.

Lemma 4.2 (Dirac [1961]). Every triangulated graph $G = (V, E)$ has a simplicial vertex. Moreover, if G is not a clique, then it has two nonadjacent simplicial vertices.

Proof. The lemma is trivial if G is complete. Assume that G has two nonadjacent vertices a and b and that the lemma is true for all graphs with fewer vertices than G. Let S be a minimal vertex separator for a and b with G_A and G_B being the connected components of G_{V-S} contain a and b, respectively.

By induction, either the subgraph G_{A+S} has two nonadjacent simplicial vertices one of which must be in A (since S induces a complete subgraph) or G_{A+S} is itself complete and any vertex of A is simplicial in G_{A+S}. Furthermore, since $Adj(A) \subseteq A+S$, a simplicial vertex of G_{A+S} in A is simplicial in all of G. Similarly B contains a simplicial vertex of G. This proves the lemma.

We now rejoin the proof of the theorem which is still in progress.

(i) $\Rightarrow$ (ii) According to the lemma, if G is triangulated, then it has a simplicial vertex, say x. Since $G_{V-\{x\}}$ is triangulated and smaller than G, it has, by induction, a perfect scheme which, when adjoined as a suffix of x, forms a perfect scheme of G.

(ii) $\Rightarrow$ (i)[4] Let C be a simple cycle of G and let x be the vertex of C with the smallest index in a perfect scheme. Since $|Adj(x) \cap C| \geq 2$, the eventual simpliciality of x guarantees a chord in C. $\square$

3. Recognizing Triangulated Graphs by Lexicographic Breadth-First Search

From Lemma 4.2 we learned that the Fulkerson-Gross recognition procedure affords us a choice of at least two vertices for each position in constructing a perfect scheme

[2] Edges xa_i, xb_i, ya_i, and yb_i are not mentioned in this proof.

[3] See our edition of this chapter for a proof of this result.

[4] See our edition for a slightly more precise proof.

(i) $\Rightarrow$ (iii) Suppose S is a minimal $a-b$ separator with G_A and G_B being the connected components of G_{V-S} containing a and b, respectively. Since S is minimal, each $x \in S$ is adjacent to some vertex in A and some vertex in B. Therefore, for any pair $x, y \in S$ there exist paths $[x, a_1, \ldots, a_r, y]$ and $[y, b_1, \ldots, b_t, x]$, where each $a_i \in A$ and $b_i \in B$, such that these paths are chosen to be of smallest possible length. It follows that $[x, a_1, \ldots, a_r, y, b_1, \ldots, b_t, x]$ is a simple cycle whose length is at least 4, implying that it must have a chord. But $a_i b_j \notin E$ by the definition of vertex separator; $xa_i \notin E$, $xb_i \notin E$, $ya_i \notin E$, $yb_i \notin E$, $a_i a_j \notin E$ and $b_i b_j \notin E$ by the minimality of r and t. Thus, the only possible chord is $xy \in E$.

Remark. It also follows that $r = t = 1$, implying that for all $x, y \in S$ there exist vertices in A and B which are adjacent to both x and y. A stronger result is given in Exercise 12.[5]

Before continuing with the remaining implications, we pause for a message from our lemma department.

Lemma 4.2 (Dirac [1961]). Every triangulated graph $G = (V, E)$ has a simplicial vertex. Moreover, if G is not a clique, then it has two nonadjacent simplicial vertices.

Proof. The lemma is trivial if G is complete. Assume that G has two nonadjacent vertices a and b and that the lemma is true for all graphs with fewer vertices than G. Let S be a minimal vertex separator for a and b with G_A and G_B being the connected components of G_{V-S} containing a and b, respectively.

By induction, either the subgraph G_{A+S} has two nonadjacent simplicial vertices one of which must be in A (since S induces a complete subgraph) or G_{A+S} is itself complete and any vertex of A is simplicial in G_{A+S}. Furthermore, since $Adj(A) \subseteq A+S$, a simplicial vertex of G_{A+S} in A is simplicial in all of G. Similarly B contains a simplicial vertex of G. This proves the lemma.

We now rejoin the proof of the theorem which is still in progress.

(i) $\Rightarrow$ (ii) According to the lemma, if G is triangulated, then it has a simplicial vertex, say x. Since $G_{V-\{x\}}$ is triangulated and smaller than G, it has, by induction, a perfect scheme which, when adjoined as a suffix of x, forms a perfect scheme of G.

(ii) $\Rightarrow$ (i) Let C be a chordless cycle of G with $|C| > 3$, σ be a perfect scheme of G, and x be the vertex of C with the smallest index in σ. Then there must exist vertices a and b such that $ax \in E$, $bx \in E$ and $ab \notin E$, since C is a cycle. However, since a and b appear after x in σ, it is also true that $ab \in E$. Contradiction. So such a C cannot exist, and this means G is triangulated. □

3. Recognizing Triangulated Graphs by Lexicographic Breadth-First Search

From Lemma 4.2 we learned that the Fulkerson-Gross recognition procedure affords us a choice of at least two vertices for each position in constructing a perfect scheme

[5] See the Addendum at the end of this chapter for a proof.

for a triangulated graph. Therefore, we can freely choose a vertex v_n to *avoid* during the whole process, saving it for the last position in a scheme. Similarly, we can pick any vertex v_{n-1} adjacent to v_n to save the $(n-1)$st position. If we continued in this manner, we would be constructing a scheme *backwards*. This is exactly what Lueker [1974] and Rose and Tarjan [1975] have done in order to give a linear-time algorithm for recognizing triangulated graphs. This version presented in Rose, Tarjan, and Lueker [1976] uses a *lexicographic* breadth-first search in which the usual queue of vertices is replaced by a queue of (unordered) subsets of the vertices which is sometimes refined but never reordered. The method (Figure 4.2) is as follows:

begin
1. assign the label $\emptyset$ to each vertex;
2. **for** $i \leftarrow n$ **to** 1 **step** - 1 **do**
3. select: pick an unnumbered vertex v with largest label;
4. $\sigma(i) \leftarrow v$; **comment** This assigns to v the number i.
5. update: **for** each unnumbered vertex $w \in Adj(v)$ **do** add i to label(w);

end

Figure A.3: Algorithm 4.1: Lex BFS.

Algorithm 4.1. Lexicographic breadth-first search.

Input: The adjacency sets of an undirected graph $G = (V, E)$.
Output: An ordering σ of the vertices.
Method: The vertices are numbered from n to 1 in the order that they are selected in line 3. This numbering fixes the position of an elimination scheme σ. For each vertex x, the *label* of x will consist of a set of numbers listed in decreasing order. The vertices can then be lexicographically ordered according to their labels. (Lexicographic order is just dictionary order, so that $9761 < 985$ and $643 < 6432$.) Ties are broken arbitrarily.

Example. We shall apply Algorithm 4.1 to the graph in Figure 4.3.[6] The vertex a is selected arbitrarily in line 3 during the first pass. The evolution of the labeling and the numbering are illustrated in Figure 4.4. Notice that the final numbering $\sigma = [c, d, e, b, a]$ is a perfect vertex elimination scheme. This is no accident.

For each value of i, let $L_i(x)$ denote the label of x when statement 4 is executed, i.e. when the ith vertex is numbered. Remember, the index is *decremented* at each successive iteration. For example, $L_n(x) = \emptyset$ for all x and $L_{n-1}(x) = \{n\}$ iff $x \in Adj(\sigma(n))$. The following properties are of prime importance:

(L1) $L_i(x) \leq L_j(x)$ if $(j \leq i)$;

(L2) $L_i(x) < L_i(y) \Rightarrow L_j(x) < L_j(y)$ if $(j < i)$;

(L3) if $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$; then there exists a vertex $d \in Adj(b) - Adj(a)$ with $\sigma^{-1}(c) < \sigma^{-1}(d)$.

[6] Edge *be* appears in the 2nd edition errata.

for a triangulated graph. Therefore, we can freely choose a vertex v_n to *avoid* during the whole process, saving it for the last position in a scheme. Similarly, we can pick any vertex v_{n-1} adjacent to v_n to save the $(n-1)$st position. If we continued in this manner, we would be constructing a scheme *backwards*. This is exactly what Lueker [1974] and Rose and Tarjan [1975] have done in order to give a linear-time algorithm for recognizing triangulated graphs. This version presented in Rose, Tarjan, and Lueker [1976] uses a *lexicographic* breadth-first search in which the usual queue of vertices is replaced by a queue of (unordered) subsets of the vertices which is sometimes refined but never reordered. The method (Figure 4.2) is as follows:

begin
1. assign the label $\emptyset$ to each vertex;
2. **for** $i \leftarrow n$ **to** 1 **step** - 1 **do**
3. select: pick an unnumbered vertex v with largest label;
4. $\sigma(i) \leftarrow v$; **comment** This assigns to v the number i.
5. update: **for** each unnumbered vertex $w \in Adj(v)$ **do** add i to label(w);

end

Figure B.4: Algorithm 4.1: Lex BFS.

Algorithm 4.1. Lexicographic breadth-first search.

Input: The adjacency sets of an undirected graph $G = (V, E)$.
Output: An ordering σ of the vertices.
Method: The vertices are numbered from n to 1 in the order that they are selected in line 3. This numbering fixes the position of an elimination scheme σ. For each vertex x, the *label* of x will consist of a set of numbers listed in decreasing order. The vertices can then be lexicographically ordered according to their labels. (Lexicographic order is just dictionary order, so that $9761 < 985$ and $643 < 6432$.) Ties are broken arbitrarily.

Example. We shall apply Algorithm 4.1 to the graph in Figure 4.3. The vertex a is selected arbitrarily in line 3 during the first pass. The evolution of the labeling and the numbering are illustrated in Figure 4.4. Notice that the final numbering $\sigma = [c, d, e, b, a]$ is a perfect vertex elimination scheme. This is no accident.

For each value of i, let $L_i(x)$ denote the label of x when statement 4 is executed, i.e. when the ith vertex is numbered. Remember, the index is *decremented* at each successive iteration. For example, $L_n(x) = \emptyset$ for all x and $L_{n-1}(x) = \{n\}$ iff $x \in Adj(\sigma(n))$. The following properties are of prime importance:

(L1) $L_i(x) \leq L_j(x)$ if $(j \leq i)$;

(L2) $L_i(x) < L_i(y) \Rightarrow L_j(x) < L_j(y)$ if $(j < i)$;

(L3) if $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$, then there exists a vertex $d \in Adj(b) - Adj(a)$ with $\sigma^{-1}(c) < \sigma^{-1}(d)$.

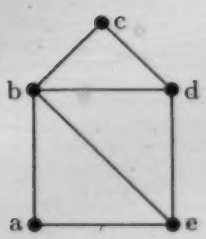

Figure A.5:

Property (L1) says that the label of a vertex may get larger but never smaller as the algorithm proceeds. Property (L2) states that once a vertex gets ahead of another vertex, they stay in that order. Finally, (L3) gives a condition under which there must be a suitable vertex d which was numbered before c (in time) and hence received a larger number.[7]

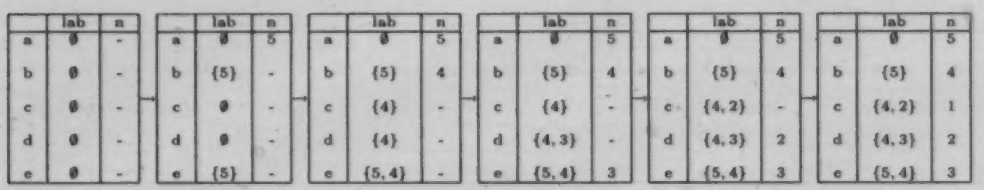

Figure A.6:

Lexicographic breadth-first search can be used to recognize triangulated graphs as demonstrated by the next theorem.

Theorem 4.3 An undirected graph $G = (V, E)$ is triangulated if and only if the ordering σ produced by Algorithm 4.1 is a perfect vertex elimination scheme.

Proof.[8] If $|V| = n = 1$, then the proof is trivial. Assume that the theorem is true for all graphs with fewer than n vertices and let σ be the ordering produced by Algorithm 4.1 when applied to a triangulated graph G. By induction, it is sufficient to show that $x = \sigma(1)$ is a simplicial vertex of G.[9]

Suppose x is not simplicial. Choose vertices x_1,$x_2 \in Adj(x)$ with $x_1x_2 \notin E$ so that x_2 is as large as possible (with respect to the ordering σ). (Remember, σ increases as you approach the root of the search tree.[10]) Consider the following inductive procedure. Assume we are given vertices $x_1, x_2, \ldots, x_m$ with these properties: for all $i, j > 0$,

[7] See our edition for the proofs of these properties.

[8] See our edition for a proof better suited for formalization.

[9] The assumption that the algorithm works on graphs with fewer than n vertices cannot be used here since we are running it on a graph which does not have fewer than n vertices.

[10] I.e, vertices numbered earlier by the algorithm are given larger numbers that those numbered later.

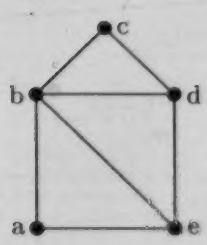

Figure B.7:

Property (L1) says that the label of a vertex may get larger but never smaller as the algorithm proceeds. Property (L2) states that once a vertex gets ahead of another vertex, they stay in that order. Finally, (L3) gives a condition under which there must be a suitable vertex d which was numbered before c (in time) and hence received a larger number.[11]

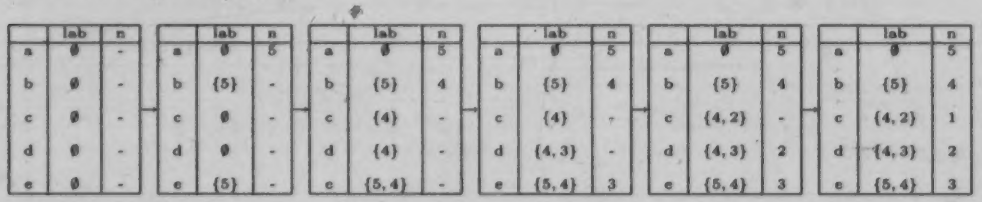

Figure B.8:

Lexicographic breadth-first search can be used to recognize triangulated graphs as demonstrated by the next theorem.

Theorem 4.3. An undirected graph $G = (V, E)$ is triangulated if and only if the ordering σ produced by Algorithm 4.1 is a perfect vertex elimination scheme.

Proof. Assume $|V| > j > 0$ steps of LexBFS have been completed (and so $i = |V| - j$ for the next step). Let V_j be the j vertices of G that have been numbered so far, G_j the subgraph of G induced by V_j, and σ_j the ordering of V_j produced by LexBFS. Assume that σ_j is a perfect vertex elimination scheme for G_j. We show that σ_{j+1} is a perfect elimination scheme for G_{j+1}.

Assume not. Then the first vertex of σ_{j+1}, x_0, is not simplicial in G_{j+1}. Choose vertices $x_1, x_2 \in Adj(x)$ with $x_1x_2 \notin E$ so that x_2 is as large as possible (with respect to the ordering σ_j). These three vertices and property L3 imply the existence of x_3, which is the largest vertex (with respect to σ_j) adjacent to x_1 and not adjacent to x_0.

[11] See the addendum for the proofs of these properties.

(1) $x, x_i \in E \Leftrightarrow i \leq 2$,

(2) $x_i x_j \in E \Leftrightarrow |i - j| = 2$,

(3) $\sigma^{-1}(x_1) < \sigma^{-1}(x_2) < \cdots < \sigma^{-1}(x_m)$,

(4) x_j is the largest vertex (with respect to σ such that $x_{j-2}x_j \in E$ but $x_{j-3}x_j \notin E$.

(For notational reasons[12] let $x_0 = x$ and $x_{-1} = x_1$.) The situation for $m = 2$ was constructed initially.

The vertices x_{m-2}, x_{m-1}, and x_m satisfy the hypothesis of property (L3) as a, b, and c respectively. Hence, choose x_{m+1} to be the largest vertex (with respect to σ) larger than x_m which is adjacent to x_{m-1} but not adjacent to x_{m-2}. Now, if x_{m+1} were adjacent to x_{m-3}, then (L3) applied to the vertices $x_{m-3}, x_{m-2}, x_{m+1}$ would imply the existence of a vertex larger than x_{m+1} (hence larger than x_m) which is adjacent to x_{m-2} but not to x_{m-3}, contradicting the maximality of x_m in (4). Therefore, x_{m+1} is not adjacent to x_{m-3}. Finally, it follows from (1),(2), and chordality that $x_i x_{m+1} \notin E$ for $i = 0, 1, \ldots, m-4, m$.

Clearly this inductive procedure continues indefinitely, but the graph is finite, a contradiction. Therefore, the vertex x must be simplicial, and the theorem is proved in one direction. The converse follows from Theorem 4.1. □

In an unpublished work, Tarjan [1976] has shown another method of searching a graph that can be used to recognize triangulated graphs. It is called *maximum cardinality search* (MCS), and it is described as follows:

MCS: The vertices are to be numbered from n to 1.
The next vertex to be numbered is always one which is adjacent to the most numbered vertices, ties being broken arbitrarily.

Using an argument similar to the proof of Theorem 4.3, one can show that G is triangulated if and only if every MCS ordering of the vertices is a perfect elimination scheme.[13] It should be pointed out that there are MCS orderings which cannot be obtained by Lex BFS, there Lex BFS orderings which are not MCS, and there exist perfect elimination schemes which are neither MCS nor Lex BFS. Exercise 27 and 28 develop some of the results on MCS. [these are lemma 2 and theorem 4.4 above] Both Lex BFS and MCS are special cases of a general method for finding perfect elimination schemes recently developed by Alan Hoffman and Michel Sakarovich.

[12] What is x_{j-3} for $j = 1$?

[13] See our edition for the proof of these results.

Now, suppose there exists an open chordless path P of G_{j+1} with $k \geq 4$ vertices such that $\sigma^{-1}(p_k) > \sigma^{-1}(p_1) > \sigma^{-1}(p_{k-1}) > \sigma^{-1}(p_2)$; for every $i \leq k$ we have $\sigma^{-1}(p_i) \leq \sigma^{-1}(p_k)$; and p_1 is the largest vertex (with respect to σ_j that is adjacent to p_2 and not adjacent to p_{k-1}. (Such a path exists; for example, x_2, x_0, x_1, x_3 meets these requirements.) Property L3 and p_{k-1}, p_1, p_k imply the existence of p_{k+1}, which is the largest vertex adjacent to p_k and not adjacent to p_2. Note that p_{k+1} is not adjacent to p_{k-1} — if it was, then L3 and p_{k-1}, p_2, p_{k+1} would give us a vertex that is larger than p_1 and adjacent to p_2 but not adjacent to p_{k-1}, a contradiction on the maximality of p_1. Since p_{k+1} is not adjacent to p_{k-1}, it follows from chordality that p_{k+1} is not adjacent to any p_i with $i < k$. We create a new path Q by appending p_{k+1} to P and then reversing this path. Q has the same properties as P that we mentioned earlier, but contains one more vertex. We can now inductively apply this procedure indefinetely. This is a contradiction, however, since G_j is a finite graph. Hence x_0 is simplicial in G_{j+1}.

We have just shown that each step of the algorithm produces a perfect vertex elimination scheme for the vertices numbered after the first j steps. Then after numbering every vertex of G we have produced a perfect vertex elimination scheme for G. The converse follows from Theorem 4.1. □

In an unpublished work, Tarjan [1976] has shown another method of searching a graph that can be used to recognize triangulated graphs. It is called *maximum cardinality search* (MCS), and it is described as follows:

MCS: The vertices are to be numbered from n to 1.
The next vertex to be numbered is always one which is adjacent to the most numbered vertices, ties being broken arbitrarily.

begin
1. assign the label 0 to each vertex;
2. **for** $i \leftarrow n$ **to** 1 **step** - 1 **do**
3. select: pick an unnumbered vertex v with largest label;
4. $\sigma(i) \leftarrow v$; **comment** This assigns to v the number i.
5. update: **for** each unnumbered vertex $w \in Adj(v)$ **do** add 1 to label(w);

end

Figure B.9: Algorithm 4.2

Using an argument similar to the proof of Theorem 4.3, one can show that G is triangulated if and only if every MCS ordering of the vertices is a perfect elimination scheme.[14] It should be pointed out that there are MCS orderings which cannot be obtained by Lex BFS, there are Lex BFS orderings which are not MCS, and there exist perfect elimination schemes which are neither MCS nor Lex BFS. Exercise 27 and 28 develop some of the results on MCS. [these are lemma 2 and theorem 4.4 above] Both Lex BFS and MCS are special cases of a general method for finding perfect elimination schemes recently developed by Alan Hoffman and Michel Sakarovich.

[14]See the Addendum for our treatment of MCS.

4. Addendum

We offer a few miscellaneous proofs in this section, as well as the proofs for the MCS algorithm.

Exercise 12. For any minimal vertex separator S of a triangulated graph $G = (V, E)$, there exists a vertex c in each connected component of G_{V-S} such that $S \subseteq Adj(c)$.

Proof. Let a and b be nonadjacent vertices of G, and let S be a nonempty minimal a-b separator. Let G_A be the connected component of G_{V-S} that contains a. Assume towards contradiction that G_A does not contain a vertex adjacent to all of S.

Take any $c \in G_A$ with maximum $|Adj(c) \cap S|$, and any $y \in S$ with $y \notin Adj(c)$. Take any shortest path between c and y in $G_A \cup \{y\}$ and call it P. Since S is minimal there is some $d \in G_A$ adjacent to y, otherwise we could remove y from S to get a smaller a-b separator. There is a path between c and d since G_A is connected. Thus the path P always exists and is given by $c \rightsquigarrow d$[15] $\rightarrow y$ (d cannot equal c because y was chosen to be nonadjacent to c and $d \in Adj(y)$). Now, it cannot be true that $(Adj(c) \cap S) \subset (Adj(d) \cap S)$ because then $|Adj(d) \cap S| > |Adj(c) \cap S|$, which would contradict the maximality of c. This means there is a vertex $x \in S$ that is adjacent to c but not adjacent to d. Consider the vertex on P that is nearest (wrt to the path order) d and also adjacent to x; call it e. Since S is a clique we know x and y are adjancent. Then $e \rightsquigarrow d \rightarrow y \rightarrow x \rightarrow e$ is a chordless cycle with length at least four, a contradiction. □

Proposition 4.9. (L1) For every vertex $x \in G$, if $j \leq i$ then $L_i(x) \leq L_j(x)$.

Proof. We show that for any $i > 0$, $L_i(x) \leq L_{i-1}(x)$; the desired result then follows by induction. So, to this end, let y be the vertex chosen to be numbered at step $i-1$. If $y \notin Adj(x)$ then $L_i(x) = L_{i-1}(x)$ and we are done. If $y \in Adj(x)$, then we add $i-1$ to $L_i(x)$ to obtain $L_{i-1}(x)$. What this really means is that we append $i-1$ to the end of $L_i(x)$, and so it impossible for $L_i(x) > L_{i-1}(x)$ to be true. This proves the result. □

Proposition 4.10. (L2) For all vertices $x, y \in G$, if $j < i$ and $L_i(x) < L_i(y)$ then $L_j(x) < L_j(y)$.

Proof. We need only show that $L_i(x) < L_i(y) \Rightarrow L_{i-1}(x) < L_{i-1}(y)$, as above. Let z be the vertex that is numbered in step $i-1$. If z is non adjacent to x and to y, then we are done since both labels remain untouched. If z is adjacent to both, then we append $i-1$ to both labels and the ordering must remain the same. If z is adjacent to y and not to x, then $L_i(y) \leq L_{i-1}(y)$ and $L_i(x) = L_{i-1}(x)$, so $L_{i-1}(x) < L_{i-1}(y)$. Finally, if z is adjacent to x and not to y, then we append $i-1$ to $L_i(x)$. This cannot make $L_{i-1}(x) > L_i(y)$ however, because there must be some value j in $L_i(y)$ large enough in order for $L_i(y) > L_i(x)$. This value remains in $L_{i-1}(y)$, and so $L_{i-1}(x) < L_{i-1}(y)$. □

[15] the notation $x \rightsquigarrow y$ refers to a chordless path of length at least one between x and y

Proposition 4.11. (L3) If $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$, then there exists a vertex $d \in Adj(b) - Adj(a)$ with $\sigma^{-1}(c) < \sigma^{-1}(d)$.

Proof. Let a, b, c be so that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$. Assume towards contradiction that for any vertex chosen before c and adjacent to b, it holds that it is also adjacent to a.

Let i be the step in which b was picked. It is then true that $\sigma^{-1}(c) \in L_i(a)$ and $\sigma^{-1}(c) \notin L_i(b)$ because c was chosen before both a and b and because c is adjacent to a and not adjacent to b. If $L_i(a)$ and $L_i(b)$ contain the same numbers greater than $\sigma^{-1}(c)$, then $L_i(a) > L_i(b)$, a contradiction (note this includes the case where $L_i(a)$ and $L_i(b)$ both contain no numbers larger than $\sigma^{-1}(c)$ as well). So there must exist some $z \in L_i(b)$ with $\sigma^{-1}(c) < z$ and $z \notin L_i(a)$ in order for $L_i(a) \leq L_i(b)$. But this implies the existence of a vertex y such that $\sigma^{-1}(y) = z$ that is adjacent to b and not to a, a contradiction. □

Definition 4.5. Let σ be an ordering of the graph $G = (V, E)$. We say that σ has property T if the following is true: for any a, b, c such that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$, there is a vertex $d \in Adj(b) - Adj(a)$ such that $\sigma^{-1}(b) < \sigma^{-1}(d)$. Note that this is very similar to property L3 above, except we only require d to be chosen before b instead of c. This can be viewed as a generalization of the $L3$ property, and in fact, the proof we give below for the correctness of MCS can be applied to LexBFS without change.

Lemma 4.6. Any ordering produced by MCS has property T.

Proof. Let σ be an ordering produced by MCS and let a, b, c be such that $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ and $c \in Adj(a) - Adj(b)$. Let X be the set of vertices chosen before b, that is, those vertices $x \in X$ such that $\sigma^{-1}(b) < \sigma^{-1}(x)$. Divide X into four disjoint sets X_1, X_2, X_3 and X_4, where X_1 contains vertices adjacent to b and not to a, X_2 contains vertices adjacent to a and not to b, X_3 contains vertices adjacent to both a and b, and X_4 the remaining vertices. We know $|X_2| \geq 1$ since $c \in X_2$.

Since $\sigma^{-1}(a) < \sigma^{-1}(b)$, b was chosen before a and so $|X_1| + |X_3| \geq |X_2| + |X_3|$. This implies that $|X_1| \geq 1$ and so the result is proved. □

Lemma 4.7. Let σ be an ordering of $G = (V, E)$. Then σ is a perfect vertex elimination scheme of G if and only if for any a, b, c such that $ab \in E$, $ac \in E$, $\sigma^{-1}(a) < \sigma^{-1}(b)$ and $\sigma^{-1}(a) < \sigma^{-1}(c)$, it holds that $bc \in E$.

Proof. For convenience, let G_x refer to the subgraph of G induced by x and all vertices following x in σ. If σ is a perfect elimination scheme, then the neighborhood of a in G_a is complete, and so $bc \in E$. Going the other way, if every such triplet has the above property, then for any vertex x it is true that x's neighborhood in G_x is complete, and hence σ is a perfect vertex elimination scheme of G.

Theorem 4.8 (Tarjan [1984]). Let $G = (V, E)$ be a chordal graph and let σ be an ordering of G. If σ has property T then σ is a perfect vertex elimination scheme.

Proof. Assume σ has property T. Consider all paths of G with at least three vertices that have a certain property Q. From these paths, let $P = (v_0, v_1, v_2, \ldots v_k)$ be the path with maximum $\sigma^{-1}(v_k)$. We define property Q as follows: for some i in the interval $[1, k-1]$, it is true that

$$\sigma^{-1}(v_0) > \sigma^{-1}(v_k) > \sigma^{-1}(v_1) > \sigma^{-1}(v_2) \cdots > \sigma^{-1}(v_i)$$

and

$$\sigma^{-1}(v_i) < \sigma^{-1}(v_{i+1}) < \cdots < \sigma^{-1}(v_k)$$

hold.

Apply property T to vertices v_1, v_k, and v_0 to obtain a vertex x such that x is adjacent to v_k and and not to v_1 and $\sigma^{-1}(v_k) < \sigma^{-1}(x)$. Since there is at least one vertex adjacent to x (ie, v_k), let $j > 1$ be the minimum such that v_j is a vertex on P that is adjacent to x (if $j = 0$ we would have a cycle of length more than three; $j \neq 1$ from property T). Now, if $\sigma^{-1}(v_0) > \sigma^{-1}(x)$, then the path $v_0, v_1, \ldots, v_j, x$ has property Q. If $\sigma^{-1}(x) > \sigma^{-1}(v_0)$, then the path $x, v_j, v_{j-1}, \ldots, v_0$ has property Q. In both cases we have a contradiction (the last vertex would be larger than v_k with respect to the ordering σ). From this we conclude that no path has property Q.

Now consider any triplet of vertices a,b,c with $ab \in E$, $ac \in E$, and so that σ orders a before both b and c. If $bc \notin E$, then one of bac or cab must have property Q. But since we have shown there cannot be any paths with property Q, it follows that no such triplets exist. Hence σ is a perfect vertex elimination scheme of G by Lemma 4.7. □

Appendix D

MIZAR abstracts

This appendix contains the MIZAR abstracts for our two source files. The first section contains the MIZAR code for the formalization of Section 3.1 and the second section contains the code for the formalization of Sections 3.3 and 3.4. These are abstracts, which means that the proofs are omitted, but the statements of all definitions and theorems are present.

1. chord.abs

```
:: Chordal Graphs
::  by Broderick Arneson and Piotr Rudnicki
::
:: Received August 18, 2006
:: Copyright (c) 2006 Association of Mizar Users

environ

 vocabularies ARYTM_1, BOOLE, CARD_1, CAT_1, ORDINAL2, FINSET_1,
      FUNCT_1, FINSEQ_1, FINSEQ_4, GRAPH_1, GLIB_000, GLIB_001, GLIB_002,
      MATRIX_2, NEWTON, XREAL_0, NAT_1, INT_1, PRE_TOPC, RELSET_1, RELAT_1,
      RELAT_2, CHORD, TOPGEN_1, SQUARE_1, GRAPH_2, MEMBERED, BHSP_3;
 notations TARSKI, XBOOLE_0, CQC_SIM1, SUBSET_1, ORDINAL1, XCMPLX_0, XXREAL_0,
      DOMAIN_1, RELAT_1, FUNCT_1, FUNCT_2, FINSEQ_1, CARD_1, FINSET_1, INT_1,
      NAT_1, GLIB_000, GLIB_001, GLIB_002, XFAMILY_1, CQC_LANG, ABIAN, ENUMSET1,
      FINSEQ_4, NUMBERS, GRAPH_2, PRE_CIRC, MEMBERED;
 constructors DOMAIN_1, NAT_1, ARYTM_2, BINARITH, FINSEQ_4, GLIB_001,
      GLIB_002, CQC_SIM1, GRAPH_2, PRE_CIRC, XXREAL_0;
 registrations RELSET_1, NAT_1, XBOOLE_0, MEMBERED, GLIB_000, FINSEQ_1,
      GLIB_001, GLIB_002, SUBSET_1, FINSET_1, INT_1, ABIAN, JORDAN1D, ORDINAL1,
      RELAT_1, PRE_CIRC, PNPROC_1, HEYTING3;
 requirements ARITHM, BOOLE, NUMERALS, REAL, SUBSET;

 begin :: Preliminaries

 theorem :: CHORD:1 :: Nat00
 for n being non zero natural number holds n-1 is natural number & 1 <= n;

 theorem :: CHORD:2 :: Nat02
 for n being odd natural number holds n-1 is natural number & 1 <= n;

 theorem :: CHORD:3 :: EvenOdd02
 for n,m being odd integer st n < m holds n <= m-2;

 theorem :: CHORD:4 :: EvenOdd03:
 for n,m being odd integer st m < n holds m+2 <= n;

 theorem :: CHORD:5 :: EvenOdd04:
 for n being odd natural number st 1 <> n
   ex m being odd natural number st m+2 = n;

 theorem :: CHORD:6 :: Odd100
 for n being odd natural number st n<>2 holds n>1;

 theorem :: CHORD:7 :: Odd101
 for n being odd natural number st n<>4 holds n>1 or n>3;

 theorem :: CHORD:8 :: Odd102
 for n being odd natural number st n<>6 holds n>1 or n>3 or n>5;

 theorem :: CHORD:9
 for n being odd natural number st n<>8 holds n>1 or n>3 or n>5 or n>7;

 theorem :: CHORD:10 :: Even100
 for n being even natural number st n<>1 holds n>0;

 theorem :: CHORD:11 :: Even101
 for n being even natural number st n<>3 holds n>0 or n>2;

 theorem :: CHORD:12 :: Even102
 for n being even natural number st n<>5 holds n>0 or n>2 or n>4;

 theorem :: CHORD:13 :: Even103
 for n being even natural number st n<>7 holds n>0 or n>2 or n>4 or n>6;

 theorem :: CHORD:14
 for p being FinSequence, n being non zero natural number
   st p is one-to-one & n <= len p holds (p.n)..p = n;

 theorem :: CHORD:15 :: Index01
 for p being non empty FinSequence, T being non empty Subset of rng p
   ex x being set st x in T & for y being set st y in T holds x..p <= y..p;

 definition let p be FinSequence, n be natural number;
   func p.followSet(n) -> finite set equals
     :: CHORD:def 1
     rng (n.len p)-cut p;
 end;

 theorem :: CHORD:16 :: Follow00
 for p being FinSequence, x being set, n being natural number
   st x in rng p & n in dom p & p is one-to-one
     holds x in p.followSet(n) iff x..p >= n;

 theorem :: CHORD:17 :: Follow03
 for p, q being FinSequence, x being set st p = <*x*>^q
 for n being non zero natural number holds p.followSet(n+1) = q.followSet(n);

 theorem :: CHORD:18 :: FinSubseq00
 for X being set, f being FinSequence of X, g being FinSubsequence of f
   st len g = len f holds g = f;

 begin :: Miscellany on graphs

 theorem :: CHORD:19 :: :: Joins01:
 for G being _Graph, S being Subset of the_Vertices_of G
 for M being inducedSubgraph of G,S
 for u,v being set st u in S & v in S
 for e being set st e Joins u,v,G holds e Joins u,v,M;

 theorem :: CHORD:20
 for G being _Graph, W being Walk of G
   holds W is Trail-like iff len W = (card W.edges())+1;

 theorem :: CHORD:21 :: Walk02
 for G being _Graph, S being Subset of the_Vertices_of G
 for M being removeVertices of G,S
 for W being Walk of G
 st (for n being odd natural number st n <= len W holds not W.n in S)
   holds W is Walk of M;

 theorem :: CHORD:22 :: Walk03
 for G being _Graph, a,b be set st a<>b
 for W being Walk of G st W.vertices() = {a,b}
   holds ex e being set st e Joins a,b,G;

 theorem :: CHORD:23 :: Walk04
```

```
for G being _Graph, S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
for W being Walk of G st W.vertices() c= S holds W is Walk of H;

theorem :: CHORD:24  :: Cyclelike01
for G1,G2 being _Graph st G1 == G2
for W1 be Walk of G1, W2 being Walk of G2
 st W1 = W2 holds W1 is Cycle-like implies W2 is Cycle-like;

theorem :: CHORD:25  :: Path01
for G being _Graph, P being Path of G, m, n being odd natural number
 st m <= len P & n <= len P & P.m = P.n
  holds m = n or (m = 1 & n = len P) or (m = len P & n = 1);

theorem :: CHORD:26
for G being _Graph, P being Path of G st P is open
for a,e,b being set
 st not a in P.vertices() & b = P.first() & e Joins a,b,G
  holds G.walkOf(a,e,b).append(P) is Path-like;

theorem :: CHORD:27  :: PathLike03
:: A similar theorem is needed for obtaining non closed Path
for G being _Graph, P,H being Path of G
 st P.edges() misses H.edges() &
    P is non trivial & P is open & H is non trivial & H is open &
    P.vertices() /\ H.vertices() = { P.first(), P.last() } &
    H.first() = P.last() & H.last() = P.first()
  holds P.append(H) is Cycle-like;

theorem :: CHORD:28  :: PathLike04
for G being _Graph, W1,W2 being Walk of G st W1.last() = W2.first()
 holds W1.append(W2).length() = W1.length() + W2.length();

theorem :: CHORD:29  :: Subgraph01
for G being _Graph, A,B being non empty Subset of the_Vertices_of G st B c= A
for H1 being inducedSubgraph of G,A
for H2 being inducedSubgraph of H1,B
  holds H2 is inducedSubgraph of G,B;

theorem :: CHORD:30  :: Subgraph01a
for G being _Graph, A,B being non empty Subset of the_Vertices_of G st B c= A
for H1 being inducedSubgraph of G,A
for H2 being inducedSubgraph of G,B
  holds H2 is inducedSubgraph of H1,B;

theorem :: CHORD:31  :: Subgraph02
for G being _Graph, S,T being non empty Subset of the_Vertices_of G st T c= S
for G2 being inducedSubgraph of G,S
  holds G2.edgesBetween(T) = G.edgesBetween(T);

:: we do not consider infinite graphs
scheme :: CHORD:sch 1
 FinGraphOrderCompInd{P[set]}:
    for G being finite _Graph holds P[G]
    provided
 for k being non zero natural number st
    (for Gk being finite _Graph st Gk.order() < k holds P[Gk]) holds
    (for Gk1 being finite _Graph st Gk1.order() = k holds P[Gk1]);

:: should be in GLIBs
theorem :: CHORD:32  :: PDistinct
for G being _Graph, W being Walk of G
 st W is open & W is Path-like holds W is vertex-distinct;

:: should be in GLIB
theorem :: CHORD:33  :: PathLike15
for G being _Graph, P being Path of G st P is open & len P > 3
for e being set
 st e Joins P.last(),P.first(),G holds P.addEdge(e) is Cycle-like;

begin :: Shortest topological path

definition let G be _Graph, W be Walk of G;
  attr W is minlength means
:: CHORD:def 2
    for W2 being Walk of G st W2 is_Walk_from W.first(),W.last()
      holds len W2 >= len W;
end;

theorem :: CHORD:34  :: WalkSubwalk00
for G being _Graph, W being Walk of G, S being Subwalk of W
 st S.first() = W.first() & S.edgeSeq() = W.edgeSeq() holds S = W;

theorem :: CHORD:35  :: LenSubwalk00
for G being _Graph, W being Walk of G, S being Subwalk of W
 st len S = len W holds S = W;

theorem :: CHORD:36
for G being _Graph, W being Walk of G st W is minlength holds W is Path like;

theorem :: CHORD:37
for G being _Graph, W being Walk of G st W is minlength holds W is Path like;

theorem :: CHORD:38  :: TopPath01
for G being _Graph, W being Walk of G
 holds (for P being Path of G
         st P is_Walk_from W.first(),W.last() holds len P >= len W)
       implies W is minlength;

theorem :: CHORD:39  :: TopPath02
for G being _Graph, W being Walk of G
  ex P being Path of G st P is_Walk_from W.first(),W.last() & P is minlength;

theorem :: CHORD:40  :: TopPath03
for G being _Graph, W being Walk of G st W is minlength
 holds for m,n being odd natural number st m+2 < n & n <= len W
        holds not ex e being set st e Joins W.m,W.n,G;

theorem :: CHORD:41  :: TopPath04
for G being _Graph, S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
for W being Walk of H st W is minlength
for m,n being odd natural number st m+2 < n & n <= len W
 holds not ex e being set st e Joins W.m,W.n,G;

theorem :: CHORD:42  :: TopPath05
for G being _Graph for W being Walk of G st W is minlength
for m,n being odd natural number st m<=n & n<=len W
```

```
  holds W.cut(m,n) is minlength;

theorem :: CHORD:43
for G being _Graph st G is connected
for A,B being non empty Subset of the_Vertices_of G st A misses B holds
  ex P being Path of G st P is minlength & P is non trivial &
  P.first() in A & P.last() in B &
  for n being odd natural number st 1 < n & n < len P
    holds not P.n in A & not P.n in B;

begin :: Adjacency and complete graphs

definition let G be _Graph, a,b be Vertex of G;
  pred a,b are_adjacent means
:: CHORD:def 3                    :: DefAdjacent
    ex e being set st e Joins a,b,G;
  symmetry;
end;

theorem :: CHORD:44  :: VAdjacent00
for G1,G2 being _Graph st G1 == G2
for u1,v1 being Vertex of G1 st u1,v1 are_adjacent
for u2,v2 being Vertex of G2 st u1=u2 & v1=v2 holds u2,v2 are_adjacent;

theorem :: CHORD:45  :: VAdjacent01
for G being _Graph, S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
for u, v being Vertex of G, t, w being Vertex of H
  st u = t & v = w holds u,v are_adjacent iff t,w are_adjacent;

theorem :: CHORD:46  :: PathLike05
for G being _Graph, W being Walk of G
 st W.first() <> W.last() & not W.first(),W.last() are_adjacent
  holds W.length() >= 2;

theorem :: CHORD:47  :: PathBuilder00
:: add sequences of vertices and edges
for G being _Graph, v1,v2,v3 being Vertex of G st
    v1<>v2 & v1<>v3 & v2<>v3 & v1,v2 are_adjacent & v2,v3 are_adjacent
  ex P being Path of G, e1,e2 being set st
     P is open & len P = 5 & P.length() = 2 &
     e1 Joins v1,v2,G & e2 Joins v2,v3,G & P.edges() = {e1,e2} &
     P.vertices() = {v1,v2,v3} & P.1 = v1 & P.3 = v2 & P.5 = v3;

theorem :: CHORD:48   :: PathBuilder01
for G being _Graph, v1,v2,v3,v4 being Vertex of G st
  v1<>v2 & v1<>v3 & v2<>v3 & v2<>v4 & v3<>v4 &
  v1,v2 are_adjacent & v2,v3 are_adjacent & v3,v4 are_adjacent
 ex P being Path of G st len P = 7 & P.length() = 3 &
     P.vertices() = {v1,v2,v3,v4} & P.1 = v1 & P.3 = v2 & P.5 = v3 & P.7 = v4;

definition let G be _Graph, S be set;
  func G.AdjacentSet(S) -> Subset of the_Vertices_of G equals
:: CHORD:def 4
      {u where u is Vertex of G :
        not u in S & ex v being Vertex of G st (v in S & u,v are_adjacent)};
end;

theorem :: CHORD:49
for G being _Graph, S, x being set st x in G.AdjacentSet(S) holds not x in S;

theorem :: CHORD:50  :: Adjacent00
for G being _Graph, S being set
for u being Vertex of G
 holds u in G.AdjacentSet(S) iff
      not u in S & ex v being Vertex of G st (v in S & u,v are_adjacent);

theorem :: CHORD:51  :: Adjacent01
for G1,G2 being _Graph st G1 == G2 for S being set
 holds G1.AdjacentSet(S) = G2.AdjacentSet(S);

theorem :: CHORD:52  :: AdjacentV00
for G being _Graph, u,v being Vertex of G
 holds u in G.AdjacentSet({v}) iff (u <> v & v,u are_adjacent);

theorem :: CHORD:53
for G being _Graph, x,y being set
 holds x in G.AdjacentSet({y}) iff y in G.AdjacentSet({x});

theorem :: CHORD:54
for G being _Graph, C being Path of G st C is Cycle-like & C.length() > 3
for x being Vertex of G st x in C.vertices()
  ex m,n being odd Nat st m+2 < n & n <= len C & not (m=1 & n = len C) &
                          not (m=1 & n = len C-2) & not (m=3 & n = len C)'&
        C.m <> C.n & C.m in G.AdjacentSet({x}) & C.n in G.AdjacentSet({x});

theorem :: CHORD:55  :: Cycle01a
for G being _Graph, C being Path of G st C is Cycle-like & C.length() > 3
for x being Vertex of G st x in C.vertices()
 ex m,n being odd natural number st m+2 < n & n <= len C &
        C.m <> C.n & C.m in G.AdjacentSet({x}) & C.n in G.AdjacentSet({x}) &
        for e being set st e in C.edges() holds not e Joins C.m,C.n,G;

:: Gilbert's definition of isolated does not allow a vertex to have a
:: loop and a vertex just with a loop on it is NOT isolated.
:: This needs to be fixed, e.g.
::       v is isolated means G.AdjacentSet({v}) = {}
:: But we can keep the old one and the new one can be expressed just
:: by G.AdjacentSet({v}) = {}.  Should this be revised?
:: Ask Lorna and Ryan.  For loopless graphs it
:: does not matter, see below.

theorem :: CHORD:56 :; AdjacentV01: :: :: AdjacentV01
for G being loopless _Graph, u being Vertex of G
 holds G.AdjacentSet({u}) = {} iff u is isolated;

theorem :: CHORD:57  :: Connected0
for G being _Graph, G0 being Subgraph of G,
    S being non empty Subset of the_Vertices_of G,
    x being Vertex of G,
    G1 being (inducedSubgraph of G,S),
    G2 being (inducedSubgraph of G,S\/{x})
 st G1 is connected & x in G.AdjacentSet(the_Vertices_of G1)
  holds G2 is connected;

theorem :: CHORD:58  :: Simplicial2a
for G being _Graph for S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
```

```
for u being Vertex of G st u in S & G.AdjacentSet({u}) c= S
for v being Vertex of H st u=v holds G.AdjacentSet({u}) = H.AdjacentSet({v});

:: Adjacency set as a subgraph of G
definition let G be _Graph, S be set;
  mode AdjGraph of G,S -> Subgraph of G means
:: CHORD:def 5
    it is inducedSubgraph of G,G.AdjacentSet(S)
    if S is Subset of the_Vertices_of G;
end;

theorem :: CHORD:59  :: AdjGraph00
for G1, G2 be _Graph st G1 == G2
for u1 being Vertex of G1, u2 being Vertex of G2 st u1 = u2
for H1 being AdjGraph of G1,{u1}, H2 being AdjGraph of G2,{u2} holds H1 == H2;

theorem :: CHORD:60  :: Simplicial2b
for G being _Graph for S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
for u being Vertex of G
  st u in S & G.AdjacentSet({u}) c= S & G.AdjacentSet({u}) <> {}
for v being Vertex of H st u=v
for Ga being AdjGraph of G,{u}, Ha being AdjGraph of H,{v} holds Ga == Ha;

definition let G be _Graph;
  attr G is complete means
:: CHORD:def 6
    for u,v being Vertex of G st u <> v holds u,v are_adjacent;
end;

theorem :: CHORD:61  :: Completetr
for G being _Graph st G is trivial holds G is complete;

registration
  cluster trivial -> complete _Graph;
end;

registration
  cluster trivial simple complete _Graph;

  cluster non trivial finite simple complete _Graph;
end;

theorem :: CHORD:62  :: Complete00
for G1,G2 being _Graph st G1 == G2 holds G1 is complete implies G2 is complete;

theorem :: CHORD:63  :: Complete01
for G being complete _Graph, S being Subset of the_Vertices_of G
for H being inducedSubgraph of G,S holds H is complete;

begin :: Simplicial vertex :: Golumbic p. 81

definition let G be _Graph, v be Vertex of G;
  attr v is simplicial means
:: CHORD:def 7
   G.AdjacentSet({v}) <> {} implies
       for G2 being AdjGraph of G,{v} holds G2 is complete;
end;

theorem :: CHORD:64  :: Simplicial0
for G being complete _Graph, v being Vertex of G holds v is simplicial;

theorem :: CHORD:65  :: Simplicial01
for G being trivial _Graph, v being Vertex of G holds v is simplicial;

theorem :: CHORD:66  :: Simplicial1
for G1,G2 being _Graph st G1 == G2
for u1 being Vertex of G1, u2 being Vertex of G2
 st u1=u2 & u1 is simplicial holds u2 is simplicial;

theorem :: CHORD:67  :: Simplicial2
for G being _Graph for S being non empty Subset of the_Vertices_of G
for H being inducedSubgraph of G,S
for u being Vertex of G st u in S & G.AdjacentSet({u}) c= S
for v being Vertex of H st u=v holds u is simplicial iff v is simplicial;

theorem :: CHORD:68  :: Simplicial03
for G being _Graph, v being Vertex of G st v is simplicial
for a,b being set st a<>b & a in G.AdjacentSet({v}) & b in G.AdjacentSet({v})
  holds ex e being set st e Joins a,b,G;

theorem :: CHORD:69  :: Simplicial03a
for G being _Graph, v being Vertex of G
 st not v is simplicial
   ex a,b being Vertex of G st a<>b & v<>a & v<>b &
          v,a are_adjacent & v,b are_adjacent & not a,b are_adjacent;

begin :: Vertex separator  :: Golumbic, p. 84

definition let G be _Graph, a,b be Vertex of G;
 assume  a<>b & not a,b are_adjacent;
 mode VertexSeparator of a,b -> Subset of the_Vertices_of G means
:: CHORD:def 8
  not a in it & not b in it &
  for G2 being removeVertices of G,it holds
     not (ex W being Walk of G2 st W is_Walk_from a,b);
end;

theorem :: CHORD:70  :: VS01
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b holds S is VertexSeparator of b,a;

:: alternate characterization of Vertex Separator
theorem :: CHORD:71  :: VS02
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being Subset of the_Vertices_of G holds
   S is VertexSeparator of a,b iff
   (not a in S & not b in S &
    for W being Walk of G st W is_Walk_from a,b holds
       ex x being Vertex of G st x in S & x in W.vertices());

theorem :: CHORD:72  :: VS07
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b
for W being Walk of G st W is_Walk_from a,b
   ex k being odd Nat st 1 < k & k < len W & W.k in S;

theorem :: CHORD:73  :: VS08a
```

```
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b
 st S = {} holds not ex W being Walk of G st W is_Walk_from a,b;

theorem :: CHORD:74
for G being _Graph, a,b being Vertex of G
 st a<>b &not a,b are_adjacent & not ex W being Walk of G st W is_Walk_from a,b
  holds {} is VertexSeparator of a,b;

theorem :: CHORD:75  :: VS11
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b, G2 being removeVertices of G,S
for a2 being Vertex of G2 st a2=a holds G2.reachableFrom(a2) /\ S = {};

theorem :: CHORD:76  :: VS11b
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b, G2 being removeVertices of G,S
for a2,b2 being Vertex of G2 st a2=a & b2=b holds
   G2.reachableFrom(a2) /\ G2.reachableFrom(b2) = {};

theorem :: CHORD:77  :: VS10
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b for G2 being removeVertices of G,S
 holds a is Vertex of G2 & b is Vertex of G2;

definition let G be _Graph, a,b be Vertex of G;
  let S be VertexSeparator of a,b;
  attr S is minimal means
:: CHORD:def 9
      for T being Subset of S st T <> S holds not T is VertexSeparator of a,b;
end;

theorem :: CHORD:78  :: VS000
for G being  _Graph, a,b being Vertex of G
for S being VertexSeparator of a,b st S = {} holds S is minimal;

theorem :: CHORD:79  :: minVSexistance
for G being finite _Graph for a,b being Vertex of G
 ex S being VertexSeparator of a,b st S is minimal;

theorem :: CHORD:80  :: VS13
:: Property "symmetry" for 2 argument modes could be used if we had it
:: as VertexSeparator of a,b is a VertexSeparator of b,a
:: then this theorem would not be needed
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for T being VertexSeparator of b,a st S=T holds T is minimal;

theorem :: CHORD:81 :: VS06: :: :: VS06
for G being _Graph, a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for x being Vertex of G st x in S
    ex W being Walk of G st W is_Walk_from a,b & x in W.vertices();

theorem :: CHORD:82  :: VertexSep0
for G being _Graph
for a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for H being removeVertices of G,S for aa being Vertex of H st aa=a
for x being Vertex of G st x in S
    ex y being Vertex of G st y in H.reachableFrom(aa) & x,y are_adjacent;

theorem :: CHORD:83  :: VertexSep01
:: Property "symmetry" for 2 argument modes could be used if we had it
:: as VertexSeparator of a,b is a VertexSeparator of b,a
for G being _Graph
for a,b being Vertex of G st a<>b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for H being removeVertices of G,S for aa being Vertex of H st aa=b
for x being Vertex of G st x in S
    ex y being Vertex of G st y in H.reachableFrom(aa) & x,y are_adjacent;

begin :: Chordal graphs :: Golumbic, p. 81

:: The notion of a chord.  Is it worthwhile having it?

:: definition let G be _Graph, W be Walk of G, e be set;
::   pred e is_chord_of W means
::    ex m, n being odd Nat st m < n & n <= len W & W.m <> W.n &
::       e Joins W.m,W.n,G &
::       for f being set st f in W.edges() holds not f Joins W.m,W.n,G;
:: end;

:: More general notion of a chordal Walk.  Is such a notion useful? Or
:: should we stick with chordal Path?

definition let G be _Graph, W be Walk of G;
  attr W is chordal means
:: CHORD:def 10
  ex m, n being odd natural number st m+2 < n & n <= len W & W.m <> W.n &
     (ex e being set st e Joins W.m,W.n,G) &
     for f being set st f in W.edges() holds not f Joins W.m,W.n,G;
end;

notation let G be _Graph, W be Walk of G;
  antonym W is chordless for W is chordal;
end;

:: The other characterization of chordal is 'more' technical and
:: sometimes more convenient to work with.  Is this really true?
:: I have tried to save as much as possible of what Broderic has already done.
:: Need separate theorems for walks and paths!  They cannot be put as an iff.

theorem :: CHORD:84  :: ChordalWalk01
for G being _Graph, W being Walk of G
 st W is chordal
  ex m,n being odd natural number st m+2 < n & n <= len W & W.m <> W.n &
     (ex e being set st e Joins W.m,W.n,G) &
     (W is Cycle-like implies  not (m=1 & n = len W) &
                               not (m=1 & n = len W-2) &
                               not (m=3 & n = len W));

theorem :: CHORD:85  :: ChordalPath01
for G being _Graph, P being Path of G
 st ex m,n being odd natural number st m+2 < n & n <= len P &
       (ex e being set st e Joins P.m,P.n,G) &
       (P is Cycle-like implies  not (m=1 & n = len P) &
                                 not (m=1 & n = len P-2) &
```

```
                                not (m=3 & n = len P))
  holds P is chordal;

theorem :: CHORD:86  :: ChordalWalk02
for G1,G2 being _Graph st G1 == G2
for W1 being Walk of G1, W2 being Walk of G2
 st W1=W2 holds W1 is chordal implies  W2 is chordal;

theorem :: CHORD:87  :: ChordalWalk03
for G being _Graph, S being non empty Subset of the_Vertices_of G,
    H being (inducedSubgraph of G,S), W1 being Walk of G, W2 being Walk of H
  st W1 = W2 holds W2 is chordal iff W1 is chordal;

theorem :: CHORD:88
for G being _Graph, W being Walk of G
 st W is Cycle-like & W is chordal & W.length()=4
   holds ex e being set st e Joins W.1,W.5,G or e Joins W.3,W.7,G;

theorem :: CHORD:89  :: MinChordal01
for G being _Graph, W being Walk of G st W is minlength holds W is chordless;

theorem :: CHORD:90
for G being _Graph, W being Walk of G
 st W is open & len W = 5 & not W.first(),W.last() are_adjecent
  holds W is chordless;

theorem :: CHORD:91
for G being _Graph, W being Walk of G
  holds W is chordal iff W.reverse() is chordal;

theorem :: CHORD:92  :: CPath03
for G being _Graph, P being Path of G st P is open & P is chordless
  for m,n being odd natural number st m < n & n <= len P holds
    (ex e being set st e Joins P.m,P.n,G) iff m=2 + n;

theorem :: CHORD:93
for G being _Graph, P being Path of G st P is open & P is chordless
for m,n being odd natural number st m < n & n <= len P
  holds P.cut(m,n) is chordless & P.cut(m,n) is open;

theorem :: CHORD:94
for G being _Graph, S being non empty Subset of the_Vertices_of G,
    H being (inducedSubgraph of G,S), W being Walk of G, V being Walk of H
 st W = V holds W is chordless iff V is chordless;

definition let G be _Graph;
 attr G is chordal means
:: CHORD:def 11
  for P being Walk of G st P.length() > 3 & P is Cycle-like holds P is chordal;
end;

theorem :: CHORD:95  :: Chordal01
for G1,G2 being _Graph st G1 == G2 holds G1 is chordal implies G2 is chordal;

theorem :: CHORD:96  :: Chordal02
for G being finite _Graph st card the_Vertices_of G <= 3 holds G is chordal;

registration
  cluster trivial finite chordal _Graph;

  cluster non trivial finite simple chordal _Graph;

  cluster complete -> chordal _Graph;
end;

registration let G be chordal _Graph, V be set;
  cluster -> chordal inducedSubgraph of G,V;
end;

theorem :: CHORD:97
for G being chordal _Graph, P being Path of G st P is open & P is chordless
for x,e being set st (not x in P.vertices() & e Joins P.last(),x,G &
                        not ex f being set st f Joins P.(len P-2),x,G)
  holds P.addEdge(e) is Path-like & P.addEdge(e) is open &
        P.addEdge(e) is chordless;


:: Golumbic, page 83, Theorem 4.1  (i) ==> (iii)
theorem :: CHORD:98  :: :: Chordal41:  : FR
for G being chordal _Graph, a,b being Vertex of G
 st a<>b & not a,b are_adjacent
  for S being VertexSeparator of a,b st S is minimal & S is non empty
  for H being inducedSubgraph of G,S holds H is complete;

:: Golumbic, page 83, Theorem 4.1 (iii) >(i)
theorem :: CHORD:99 ::: DiracThm2: :: :: DiracThm2
for G being finite _Graph
 st for a,b being Vertex of G st a<>b & not a,b are_adjacent
    for S being VertexSeparator of a,b st S is minimal & S is non empty
    for G2 being inducedSubgraph of G,S holds G2 is complete
  holds G is chordal;

:: Exercise 12, p. 101
:: This needs "finite branching", we do it for finite though
theorem :: CHORD:100  :: tExercise12
for G being finite chordal _Graph, a, b being Vertex of G
 st a <> b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for H being removeVertices of G,S, a1 being Vertex of H st a = a1
 ex c being Vertex of G st c in H.reachableFrom(a1) &
   for x being Vertex of G st x in S holds c,x are_adjacent;

theorem :: CHORD:101 :: Remark p. 83
for G being finite chordal _Graph, a, b being Vertex of G
 st a <> b & not a,b are_adjacent
for S being VertexSeparator of a,b st S is minimal
for H being removeVertices of G,S, a1 being Vertex of H st a = a1
for x, y being Vertex of G st x in S & y in S holds
 ex c being Vertex of G
  st c in H.reachableFrom(a1) & c,x are_adjacent & c,y are_adjacent;

:: Golumbic, page 83, Lemma 4.2.
theorem :: CHORD:102  :: DiracLemma2
for G being non trivial finite chordal _Graph
 st not G is complete
  ex a,b being Vertex of G
   st a<>b & not a,b are_adjacent & a is simplicial & b is simplicial;
```

```
theorem :: CHORD:103  :: DiracLemma1
for G being finite chordal _Graph ex v being Vertex of G st v is simplicial;

begin :: Vertex Elimination Scheme :: Golumbic, p. 83

definition let G be finite _Graph;
  mode VertexScheme of G -> FinSequence of the_Vertices_of G means
:: CHORD:def 12
    it is one-to-one & rng it = the_Vertices_of G;
end;

registration
  let G be finite _Graph;
  cluster -> non empty VertexScheme of G;
end;

theorem :: CHORD:104
for G being finite _Graph, S being VertexScheme of G
 holds len S = card the_Vertices_of G;

theorem :: CHORD:105
for G being finite _Graph, S being VertexScheme of G holds 1 <= len S;

theorem :: CHORD:106
for G, H being finite _Graph, g being VertexScheme of G
 st G == H holds g is VertexScheme of H;

definition let G be finite _Graph, S be VertexScheme of G, x be Vertex of G;
  redefine func x..S  -> non zero Element of NAT;
end;

definition let G be finite _Graph, S be VertexScheme of G, n be Nat;
  redefine func S.followSet(n)  -> Subset of the_Vertices_of G;
end;

theorem :: CHORD:107  :: NeVSchFol:
for G being finite _Graph, S being VertexScheme of G,
    n being non zero natural number st n <= len S
  holds S.followSet(n) is non empty;

definition let G be finite _Graph, S be VertexScheme of G;
  attr S is perfect means
:: CHORD:def 13
    for n being non zero natural number st n <= len S
    for Gf being inducedSubgraph of G,S.followSet(n)
    for v being Vertex of Gf st v = S.n holds v is simplicial;
end;

:: finite is needed unless we add loopless
theorem :: CHORD:108  :: PerScheme01
for G being finite trivial _Graph, v being Vertex of G
  ex S being VertexScheme of G st S = <*v*> & S is perfect;

theorem :: CHORD:109
for G being finite _Graph, V being VertexScheme of G holds
  V is perfect iff
  for a,b,c being Vertex of G st b<>c & a,b are_adjacent & a,c are_adjacent
  for va,vb,vc being natural number
    st va in dom V & vb in dom V & vc in dom V & V.va = a &
       V.vb = b & V.vc = c & va < vb & va < vc
    holds b,c are_adjacent;

:: Golumbic pg 83-84, Theorem 4.1 (i) ==> (ii)
registration let G be finite chordal _Graph;
  cluster perfect VertexScheme of G;
end;

theorem :: CHORD:110
for G, H being finite chordal _Graph, g being perfect VertexScheme of G
 st G == H holds g is perfect VertexScheme of H;

:: Golumbic pg 83-84,  Theorem 4.1 (ii) ==> (i)
theorem :: CHORD:111 :: Chordal4lc:
for G being finite _Graph
 st ex S being VertexScheme of G st S is perfect holds G is chordal;
```

2. lexbfs.abs

```
:: Recognizing chordal graphs: Lex BFS and MCS
:: Broderick Arneson and Piotr Rudnicki
:: August 2006

environ

 vocabularies ARYTM_1, ARYTM, BOOLE, CARD_1, CAT_1, FINSEQ_1,
      FINSET_1, FUNCOP_1, FUNCT_1, FUNCT_4, GRAPH_1, GLIB_000, GLIB_001,
      GLIB_002, GLIB_003, MATRIX_2, MEASURE2, NEWTON, NUMBERS, ORDINAL1,
      ORDINAL2, PARTFUN1, PBOOLE, POLYNOM1, PRE_TOPC, SQUARE_1, SUBSET,
      REALSET1, RELAT_1, RELAT_2, SEQM_3, CHORD, BAGORDER, UPROOTS, TOPGEN_1,
      FINSEQ_4, NAT_1, ARYTM_3, WELLORD1, ALGSEQ_1, LEXBFS, CARD_FIL;
 notations TARSKI, XBOOLE_0, ZFMISC_1, NUMBERS, SUBSET_1, XREAL_0, XREAL_1,
      RELAT_1, FUNCT_2, WELLORD1, MEMBERED, PARTFUN1, FUNCT_1, FUNCT_2,
      BINARITH, PBOOLE, ORDINAL1, CARD_1, SEQM_3, POLYNOM1, FINSET_1, XCMPLX_0,
      NAT_1, CQC_LANG, FUNCT_4, GLIB_000, GLIB_001, GLIB_002, PRE_CIRC,
      GLIB_003, BAGORDER, TERMORD, UPROOTS, CHORD, FINSEQ_1, FINSEQ_4,
      DOMAIN_1, ABIAN, RELSET_1, CARD_FIL;
 constructors DOMAIN_1, ARYTM_3, UPROOTS, BAGORDER, TERMORD, GLIB_002,
      GLIB_003, CHORD, WELLORD1, PRE_CIRC, XREAL_0, CARD_FIL;
 registrations RELSET_1, FINSET_1, NAT_1, INT_1, GLIB_000, GLIB_001,
      GLIB_002, GLIB_003, CARD_FIL, CHORD, POLYNOM1, FINSEQ_1, FUNCT_1,
      XREAL_0, TERMORD, BAGORDER, XBOOLE_0, ABIAN, PRE_CIRC, MEMBERED,
      ORDINAL1, HEYTING3, XREAL_0;
 requirements NUMERALS, SUBSET, BOOLE, REAL, ARITHM;
 definitions GLIB_000, GLIB_001, GLIB_002, GLIB_003, FINSEQ_1;
 theorems ARYTM_0, CARD_1, CARD_2, CARD_4, CQC_LANG, FINSEQ_1, FINSEQ_2,
      FINSEQ_3, FINSET_1, FUNCOP_1, FUNCT_1, FUNCT_2, FUNCT_4, GLIB_000,
      GLIB_001, GLIB_002, GLIB_003, HEYTING3, INT_1, NAT_1, BAGORDER,
      TERMORD, ORDINAL1, PARTFUN1, PBOOLE, REAL_1, RELAT_1, RELSET_1,
      SCM_1, TARSKI, XBOOLE_0, XBOOLE_1, XREAL_1, ZFMISC_1, PEPIN, SRINGS1,
      SEQM_3, CHORD, BINARITH, RECDEF_2, WELLORD1, UPROOTS,
      BAGORDER, CARD_FIL, POLYNOM1, ORDINAL3, TREES_1, PRE_CIRC, MEMBERED,
      NAT_2;
 schemes BINARITH, NAT_1, FUNCT_1, RECDEF_1, GOBOARD1, FRAENKEL;

begin :: Preliminaries

 :: Nore general than GRAPH_2:4

 theorem :: LEXBFS:1
 for A,B being Element of NAT, X being non empty set
 for F being Function of NAT, X st F is one-to-one
   holds Card (F.:(a qua Element of NAT: A <= a & a <= A + B) + B+1);

 theorem :: LEXBFS:2
 for n,m,k being natural number st m <= k & n < m holds k - m < k - n;

 theorem :: LEXBFS:3
 for n,k being natural number st n < k holds k - (n+1) + 1 = k - n;

 theorem :: LEXBFS:4 :: Div00
 for n,m,k being natural number st k <> 0 holds (n + m*k) div k = (n div k) + m;

 definition let S be set;
   attr S is with_finite_elements means
 :: LEXBFS:def 1
     for x being Element of S holds x is finite;
 end;

 registration
   cluster non empty with_finite_elements set;
   cluster non empty finite with_finite_elements Subset of bool NAT;
 end;

 registration let S be with_finite_elements set;
   cluster -> finite Element of S;
 end;

 definition let f,g be Function;
   func f .\/ g -> Function means
 :: LEXBFS:def 2
     dom it = dom f \/ dom g &
     for x being set st x in dom f \/ dom g holds it.x = f.x \/ g.x;
 end;

 theorem :: LEXBFS:5
 for n,k being natural number holds
   n in ((Seg k) \ Seg (k -' n)) iff k -' n < n & n <= k;

 theorem :: LEXBFS:6
 for n,k,m being natural number st n <= m holds
   ((Seg k) \ Seg (k -' n)) c= ((Seg k) \ Seg (k -' m));

 theorem :: LEXBFS:7
 for n,k being natural number st n < k holds
   ((Seg k) \ Seg (k -' n)) \/ (k -' n) = ((Seg k) \ Seg (k -' (n+1)));

 definition let f be Relation;
   attr f is natsubset-yielding means
 :: LEXBFS:def 3
     rng f c= bool NAT;
 end;

 registration
   cluster finite-yielding natsubset yielding Function;
 end;

 definition let f be finite-yielding natsubset-yielding Function, x be set;
   redefine func f.x -> finite Subset of NAT;
 end;

 theorem :: LEXBFS:8
 for X being Ordinal, a, b be finite Subset of X st a <> b
   holds (a.:) hag <> (b.:) hag;

 definition let F be natural-yielding Function, S be set, k be natural number;
   func F.:^subset(S,k) -> natural-yielding Function means
 :: LEXBFS:def 4
     dom it = dom F &
     for y being set holds (y in S & y in dom F implies it.y = F.y + k) &
                           (not y in S implies it.y = F.y);
 end;

 definition let a be Ordinal, T be connected TermOrder of a,
                B be non empty finite Subset of Segm n;
   func max(B,T) -> Segm n means
 :: LEXBFS:def 5
     it in B & for y being Segm n st y in B holds [y,it] in T or y = it;
```

```
    it in B & for x being beg of n st x in B holds x <= it.T;
end;

registration let G be Ordinal;
  cluster InvLexOrder G -> connected;
end;

begin :: Miscellany on graphs

registration let G be _Graph;
  cluster non empty one-to-one VertexSeq of G;
end;

definition
  let G be _Graph, V being non empty VertexSeq of G;
  mode Walk of V -> Walk of G means
:: LEXBFS:def 5
  it.vertexSeq() = V;
end;

registration
  let G be _Graph, V being non empty one-to-one VertexSeq of G;
  cluster -> Path-like Walk of V;
end;

theorem :: LEXBFS:9 :: PathLike20
for G being _Graph, W1,W2 being Walk of G
 st W1 is trivial & W1.last() = W2.first() holds W1.append(W2) = W2;

theorem :: LEXBFS:10
for G, H being _Graph, A, B, C being set,
    G1 being (inducedSubgraph of G,A), H1 being (inducedSubgraph of H,B),
    G2 being (inducedSubgraph of G1,C), H2 being (inducedSubgraph of H1,C)
 st G == H & A c= B & C c= A & C is non empty Subset of the_Vertices_of G
  holds G2 == H2;

definition let G be VGraph;
  attr G is natural-vlabeled means
:: LEXBFS:def 7
    the_VLabel_of G is natural-yielding;
end;

begin :: Graphs with two vertex labels

definition
  func V2LabelSelector -> natural number equals
:: LEXBFS:def 8
0;
end;

definition let G be GraphStruct;
  attr G is [V2Labeled] means
:: LEXBFS:def 9  ::dV2LABELED
    V2LabelSelector in dom G &
    ex f being Function st G.V2LabelSelector = f & dom f c= the_Vertices_of G;
end;

registration
  cluster [Graph-like] [Weighted] [ELabeled] [VLabeled] [V2Labeled]
          GraphStruct;
end;

definition
  mode V2Graph is [V2Labeled] _Graph;
  mode VVGraph is [VLabeled] [V2Labeled] _Graph;
end;

definition let G be V2Graph;
  func the_V2Label_of G -> Function equals
:: LEXBFS:def 10
G.V2LabelSelector;
end;

theorem :: LEXBFS:11
for G being V2Graph holds dom the_V2Label_of G c= the_Vertices_of G;

registration let G be _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> [Graph-like];
end;

theorem :: LEXBFS:12
for G being _Graph, X being set holds G.set(V2LabelSelector, X) == G;

registration let G be finite _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> finite;
end;

registration let G be loopless _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> loopless;
end;

registration let G be trivial _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> trivial;
end;

registration let G be non trivial _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> non trivial;
end;

registration let G be non-multi _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> non multi;
end;

registration let G be non Dmulti _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> non Dmulti;
end;

registration let G be connected _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> connected;
end;

registration let G be acyclic _Graph, X be set;
  cluster G.set(V2LabelSelector, X) -> acyclic;
end;

registration let G be VGraph, X be set;
  cluster G.set(V2LabelSelector, X) -> [VLabeled];
end;
```

```
registration let G be EGraph, X be set;
  cluster G.set(V2LabelSelector, X) -> [ELabeled];
end;

registration let G be WGraph, X be set;
  cluster G.set(V2LabelSelector, X) -> [Weighted];
end;

registration let G be V2Graph, X be set;
  cluster G.set(VLabelSelector, X) -> [V2Labeled];
end;

registration let G be _Graph, Y be set, X be PartFunc of the_Vertices_of G,Y;
  cluster G.set(V2LabelSelector, X) -> [V2Labeled];
end;

registration let G be _Graph, X be ManySortedSet of the_Vertices_of G;
  cluster G.set(V2LabelSelector, X) -> [V2Labeled];
end;

registration let G be _Graph;
  cluster G.set(V2LabelSelector, {}) -> [V2Labeled];
end;

definition let G be V2Graph;
  attr G is natural-v2labeled means
:: LEXBFS:def 11
    the_V2Label_of G is natural-yielding;
  attr G is finite-v2labeled means
:: LEXBFS:def 12
    the_V2Label_of G is finite-yielding;
  attr G is natsubset-v2labeled means
:: LEXBFS:def 13
    the_V2Label_of G is natsubset-yielding;
end;

registration
  cluster finite natural-vlabeled finite-v2labeled natsubset-v2labeled chordal
          ([Weighted] [ELabeled] [VLabeled] [V2Labeled] _Graph);

  cluster finite natural-vlabeled natural-v2labeled chordal
          ([Weighted] [ELabeled] [VLabeled] [V2Labeled] _Graph);
end;

registration let G be natural-vlabeled VGraph;
  cluster the_VLabel_of G -> natural-yielding;
end;

registration let G be natural-v2labeled V2Graph;
  cluster the_V2Label_of G -> natural-yielding;
end;

registration let G be finite-v2labeled V2Graph;
  cluster the_V2Label_of G -> finite-yielding;
end;

registration let G be natsubset-v2labeled V2Graph;
  cluster the_V2Label_of G -> natsubset-yielding;
end;

registration let G be VVGraph, v,x be set;
  cluster G.labelVertex(v,x) -> [V2Labeled];
end;

theorem :: LEXBFS:13
for G being VVGraph, v,x being set
 holds the_V2Label_of G = the_V2Label_of G.labelVertex(v,x);

registration
  let G be natural-vlabeled VVGraph, v be set, x be natural number;
  cluster G.labelVertex(v,x) -> natural-vlabeled;
end;

registration
  let G be natural-v2labeled VVGraph, v be set, x be natural number;
  cluster G.labelVertex(v,x) -> natural-v2labeled;
end;

registration
  let G be finite-v2labeled VVGraph, v be set, x be natural number;
  cluster G.labelVertex(v,x) -> finite-v2labeled;
end;

registration
  let G be natsubset-v2labeled VVGraph, v be set, x be natural number;
  cluster G.labelVertex(v,x) -> natsubset-v2labeled;
end;

:: Subgraphs and inheritence

registration let G be _Graph;
  cluster [VLabeled] [V2Labeled] Subgraph of G;
end;

definition let G be V2Graph, G2 be [V2Labeled] Subgraph of G;
  attr G2 is v2label-inheriting means
:: LEXBFS:def 14
    the_V2Label_of G2 = (the_V2Label_of G) | the_Vertices_of G2;
end;

registration let G be V2Graph;
  cluster v2label-inheriting ([V2Labeled] Subgraph of G);
end;

definition let G be V2Graph;
  mode V2Subgraph of G is v2label-inheriting ([V2Labeled] Subgraph of G);
end;

registration let G be VVGraph;
  cluster vlabel-inheriting v2label-inheriting
          ([VLabeled] [V2Labeled] Subgraph of G);
end;

definition let G be VVGraph;
  mode VVSubgraph of G is vlabel-inheriting v2label-inheriting
                          ([VLabeled] [V2Labeled] Subgraph of G);
end;
```

```
registration let G be natural-vlabeled VGraph;
  cluster -> natural-vlabeled VSubgraph of G;
end;

registration
  let G be _Graph, V,E be set;
 cluster [Weighted] [ELabeled] [VLabeled] [V2Labeled] inducedSubgraph of G.V,E;
end;

registration
  let G be VVGraph, V, E being set;
  cluster vlabel-inheriting v2label-inheriting
     ([VLabeled] [V2Labeled] inducedSubgraph of G.V,E);
end;

definition let G be VVGraph, V,E be set;
  mode inducedVVSubgraph of G.V,E is vlabel-inheriting v2label-inheriting
       ([VLabeled] [V2Labeled] inducedSubgraph of G.V,E);
end;

definition let G be VVGraph, V be set;
  mode inducedVVSubgraph of G.V is inducedVVSubgraph of G.V,G.edgesBetween(V);
end;

begin :: More on Graph Sequences

::::::::::::: this should go into glib
definition let s be ManySortedSet of NAT;
  attr s is iterative means
:: LEXBFS:def 15                    ::dGSITERATIVE:
    for k, n being natural number st s.k = s.n holds s.(k+1) = s.(n+1);
end;

definition let GS be ManySortedSet of NAT;
  attr GS is eventually-constant means
:: LEXBFS:def 16
    en n being natural number
     st for m being natural number st n <= m holds GS.n = GS.m;
end;

registration
  cluster halting iterative eventually-constant ManySortedSet of NAT;
end;

theorem :: LEXBFS:14
for Gs being ManySortedSet of NAT
  st Gs is halting & Gs is iterative holds Gs is eventually-constant;

registration
  cluster halting iterative -> eventually-constant ManySortedSet of NAT;
end;

theorem :: LEXBFS:15
for Gs being ManySortedSet of NAT
  st Gs is eventually-constant holds Gs is halting;

registration
  cluster eventually-constant  -> halting ManySortedSet of NAT;
end;

theorem :: LEXBFS:16
for Gs being iterative eventually-constant ManySortedSet of NAT
for n being natural number st Gs.Lifespan() <= n
  holds Gs.(Gs.Lifespan()) = Gs.n;

theorem :: LEXBFS:17
for Gs being iterative eventually-constant ManySortedSet of NAT
for n,m being natural number st Gs.Lifespan() <= n & n <= m
  holds Gs.m = Gs.n;

:::::::::::::::::::::::: stuff needed here
definition let GS be VGraphSeq;
  attr GS is natural vlabeled means
:: LEXBFS:def 17                    ::dGSEQVNATVL:
    for x being natural number holds GS.x is natural-vlabeled;
end;

definition let GS be GraphSeq;
  attr GS is chordal means
:: LEXBFS:def 18                              ::dGSEQCHORDAL
    for x being natural number holds GS.x is chordal;
  attr GS is fixed-vertices means
:: LEXBFS:def 19
    for n,m being natural number holds
       (the_Vertices_of (GS.n)) = (the_Vertices_of (GS.m));
  attr GS is [V2Labeled] means
:: LEXBFS:def 20                          ::dGSEQV2LABEL
    for x being natural number holds GS.x is [V2Labeled];
end;

registration
  cluster [Weighted] [ELabeled] [VLabeled] [V2Labeled] GraphSeq;
end;

definition
  mode V2GraphSeq is [V2Labeled] GraphSeq;
  mode VVGraphSeq is [VLabeled] [V2Labeled] GraphSeq;
end;

registration let GSq be V2GraphSeq, x be natural number;
  cluster GSq.x -> [V2Labeled] _Graph;
end;

definition let GSq be V2GraphSeq;
  attr GSq is natural-v2labeled means
:: LEXBFS:def 21                    ::dGSEQNATV2L
    for x being natural number holds GSq.x is natural-v2labeled;
::    attr GSq is finite-natsubset-v2labeled means      :dGSEQNATSUB: ::dGSEQNATS
UB
::       for x being natural number holds GSq.x is finite natsubset v2labeled;
  attr GSq is finite-v2labeled means
:: LEXBFS:def 22       ::dGSEQNATSUB
    for x being natural number holds GSq.x is finite v2labeled;
  attr GSq is natsubset-v2labeled means
:: LEXBFS:def 23       ::dGSEQNATSUB
    for x being natural number holds GSq.x is natsubset v2labeled;
```

```
end;

registration
  cluster finite natural-vlabeled finite-v2labeled natsubset-v2labeled chordal
          ((Weighted) [ELabeled] [VLabeled] [V2Labeled] GraphSeq);

  cluster finite natural-vlabeled natural-v2labeled chordal
          ((Weighted) [ELabeled] [VLabeled] [V2Labeled] GraphSeq);
end;

definition let Gs be VGraphSeq, x be natural number;
  redefine func Gs.x -> VGraph;
end;

registration let GSq be natural-vlabeled VGraphSeq, x be natural number;
  cluster GSq.x -> natural-vlabeled VGraph;
end;

registration let GSq be natural-v2labeled V2GraphSeq, x be natural number;
  cluster GSq.x -> natural-v2labeled V2Graph;
end;

registration
  let GSq be finite-v2labeled V2GraphSeq, x be natural number;
  cluster GSq.x -> finite-v2labeled V2Graph;
end;

registration
  let GSq be natsubset-v2labeled V2GraphSeq, x be natural number;
  cluster GSq.x -> natsubset-v2labeled V2Graph;
end;

registration let GSq be chordal GraphSeq, x be natural number;
  cluster GSq.x -> chordal _Graph;
end;

definition let Gs be VGraphSeq, n being natural number;
  redefine func Gs.n -> VGraph;
end;

registration let Gs be finite VGraphSeq, n be natural number;
  cluster Gs.n -> finite VGraph;
end;


definition let Gs be VVGraphSeq, n being natural number;
  redefine func Gs.n -> VVGraph;
end;

registration let Gs be finite VVGraphSeq, n be natural number;
  cluster Gs.n -> finite VVGraph;
end;

registration let Gs be chordal VVGraphSeq, n be natural number;
  cluster Gs.n -> chordal VVGraph;
end;

registration let Gs be natural-vlabeled VVGraphSeq, n be natural number;
  cluster Gs.n -> natural-vlabeled VVGraph;
end;

registration let Gs be finite-v2labeled VVGraphSeq,
                 n be natural number;
  cluster Gs.n -> finite-v2labeled VVGraph;
end;

registration let Gs be natsubset-v2labeled VVGraphSeq,
                 n be natural number;
  cluster Gs.n -> natsubset-v2labeled VVGraph;
end;

registration let Gs be natural-v2labeled VVGraphSeq, n be natural number;
  cluster Gs.n -> natural-v2labeled VVGraph;
end;

begin :: Vertices numbering sequences

:: FIXME: can you think of better names for these?
definition let GS be VGraphSeq;

  attr GS is vlabel initially-empty means
:: LEXBFS:def 24   ::dVLINITEMPTY
    the_VLabel_of (GS.0) = {};

  attr GS is adds-one-at-a-step means
:: LEXBFS:def 25  ::dVLMEMBERSONE
    for n being natural number st n < GS.Lifespan() holds
     (ex w being set st not w in dom (the_VLabel_of (GS.n)) &
       the_VLabel_of (GS.(n+1)) =
       (the_VLabel_of (GS.n)) +* (w --> (GS.Lifespan() 'n))));
end;

definition let GS be VGraphSeq;
  attr GS is vlabel-numbering means
:: LEXBFS:def 26
  GS is iterative
        eventually-constant
        finite
        fixed-vertices
        natural-vlabeled
        vlabel-initially-empty
        adds-one-at-a-step;
end;

registration
  cluster iterative eventually-constant finite
          fixed-vertices natural-vlabeled vlabel-initially-empty
          adds-one-at-a-step VGraphSeq;
end;

registration
  cluster vlabel-numbering VGraphSeq;
end;

registration
  cluster vlabel-numbering -> iterative VGraphSeq;
  cluster vlabel-numbering -> eventually-constant VGraphSeq;
  cluster vlabel-numbering -> finite VGraphSeq;
```

```
  cluster vlabel-numbering -> fixed-vertices VGraphSeq;
  cluster vlabel-numbering -> natural-vlabeled VGraphSeq;
  cluster vlabel-numbering -> vlabel-initially-empty VGraphSeq;
  cluster vlabel-numbering -> adds-one-at-a-step VGraphSeq;
end;

definition
  mode VLabelNumberingSeq is vlabel-numbering VGraphSeq;
end;

definition let GS be VLabelNumberingSeq, n be natural number;
  func GS.PickedAt(n) -> set means
:: LEXBFS:def 27
    it = choose (the_Vertices_of (GS.0)) if n >= GS.Lifespan() otherwise
    not it in dom (the_VLabel_of (GS.n)) & the_VLabel_of (GS.(n+1)) =
        (the_VLabel_of (GS.n)) +* (it .--> (GS.Lifespan()-'n));
end;

theorem :: LEXBFS:18
for GS being VLabelNumberingSeq,
    n being natural number st n < GS.Lifespan()
  holds GS.PickedAt(n) in (GS.(n+1)).labeledV() &
        (GS.(n+1)).labeledV() = (GS.n).labeledV() \/ {GS.PickedAt(n)};

theorem :: LEXBFS:19
for GS being VLabelNumberingSeq, n
    being natural number st n < GS.Lifespan()
  holds (the_VLabel_of (GS.(n+1))).(GS.PickedAt(n)) = GS.Lifespan()-'n;

theorem :: LEXBFS:20
for GS being VLabelNumberingSeq, n being natural number
 st n <= GS.Lifespan() holds card ((GS.n).labeledV()) = n;

theorem :: LEXBFS:21
for GS being VLabelNumberingSeq, n being natural number holds
  rng the_VLabel_of (GS.n) = (Seg GS.Lifespan()) \ Seg (GS.Lifespan()-'n);

theorem :: LEXBFS:22
for GS being VLabelNumberingSeq, n being natural number, x being set
  holds (the_VLabel_of (GS.n)).x <= GS.Lifespan() &
        ((x in (GS.n).labeledV()) implies 1 <= (the_VLabel_of (GS.n)).x);

theorem :: LEXBFS:23
for GS being VLabelNumberingSeq, n,m being natural number
 st GS.Lifespan() -' n < m & m <= GS.Lifespan()
  ex v being Vertex of GS.n
   st v in (GS.m).labeledV() & (the_VLabel_of (GS.n)).v = m;

theorem :: LEXBFS:24
for GS being VLabelNumberingSeq, m,n being natural number st m <= n
  holds the_VLabel_of (GS.m) c= the_VLabel_of (GS.n);

theorem :: LEXBFS:25
for GS being VLabelNumberingSeq, n being natural number
 holds the_VLabel_of (GS.n) is one-to-one;

theorem :: LEXBFS:26
for GS being VLabelNumberingSeq, m,n being natural number
for v being set st v in (GS.m).labeledV() & v in (GS.n).labeledV()
  holds (the_VLabel_of (GS.m)).v = (the_VLabel_of (GS.n)).v;

theorem :: LEXBFS:27
for GS being VLabelNumberingSeq, v being set, m,n being natural number
 st (v in (GS.m).labeledV() & (the_VLabel_of (GS.m)).v = n)
  holds GS.PickedAt(GS.Lifespan()-'n) = v;

theorem :: LEXBFS:28
for GS being VLabelNumberingSeq, m,n being natural number
 st n < GS.Lifespan() & n < m
   holds GS.PickedAt(n) in (GS.m).labeledV() &
         (the_VLabel_of (GS.m)).(GS.PickedAt(n)) = GS.Lifespan()  ' n;

:: Inequalities relating the vlabel and the current iteration
theorem :: LEXBFS:29
for GS being VLabelNumberingSeq, m being natural number, v being set
  st v in (GS.m).labeledV()
   holds GS.Lifespan() -' (the_VLabel_of (GS.m)).v < m &
         GS.Lifespan() -' m < (the_VLabel_of (GS.m)).v;

:: If a vertex has a larger vlabel than we do at some point in the
:: algorithm, then it must have been in the vlabel when we were picked
theorem :: LEXBFS:30
for GS being VLabelNumberingSeq
for i being natural number, a,b being set
 st a in (GS.i).labeledV() & b in (GS.i).labeledV() &
    (the_VLabel_of (GS.i)).a < (the_VLabel_of (GS.i)).b
  holds b in (GS.(GS.Lifespan()  ' (the_VLabel_of (GS.i)).a)).labeledV();

theorem :: LEXBFS:31
for GS being VLabelNumberingSeq
for i being natural number, a,b being set
 st a in (GS.i).labeledV() & b in (GS.i).labeledV() &
    (the_VLabel_of (GS.i)).a < (the_VLabel_of (GS.i)).b
 holds not a in (GS.(GS.Lifespan() -' (the_VLabel_of (GS.i)).b)).labeledV();

begin :: Lexicographical Breadth First Search

definition let G be _Graph;
  func LexBFS:Init(G) ->
                  natural-vlabeled finite-v2labeled natsubset v2labeled
                  VVGraph equals
:: LEXBFS:def 28
    G.set(VLabelSelector, {}).set(V2LabelSelector, the_Vertices_of G --> {});
end;

definition let G be finite _Graph;
  redefine func LexBFS:Init(G)
              -> finite natural-vlabeled
                finite-v2labeled natsubset-v2labeled VVGraph;
end;

definition let G be finite finite-v2labeled natsubset-v2labeled VVGraph;
  assume  dom the_V2Label_of G = the_Vertices_of G;
  func LexBFS:PickUnnumbered(G) -> Vertex of G means
:: LEXBFS:def 29
  it = choose the_Vertices_of G if dom the_VLabel_of G = the_Vertices_of G
  otherwise
  ex S being non empty finite Subset of bool NAT,
```

```
    B being non empty finite Subset of Bags NAT,
    F being Function
  st S = rng F &
    F = ((the_V2Label_of G) | (the_Vertices_of G \ dom the_VLabel_of G)) &
    (for x being finite Subset of NAT holds x in S implies ((x,1)-bag in B)) &
    (for x being set holds x in B implies
          ex y being finite Subset of NAT`st y in S & x = (y,1)-bag) &
   it = choose (F * (support max(B,InvLexOrder NAT)));
end;

definition let G be VVGraph, v be set, k be natural number;
 func LexBFS:LabelAdjacent(G, v, k) -> VVGraph equals
:: LEXBFS:def 30
 G.set(V2LabelSelector, (the_V2Label_of G) .\/
                        ((G.AdjacentSet((v)) \ dom the_VLabel_of G)-->{k}));
end;

theorem :: LEXBFS:32
for G being VVGraph, v,x being set, k being natural number
 st not x in G.AdjacentSet((v))
  holds (the_V2Label_of G).x = (the_V2Label_of LexBFS:LabelAdjacent(G,v,k)).x;

theorem :: LEXBFS:33
for G being VVGraph, v,x being set, k being natural number
 st x in dom (the_VLabel_of G) holds
   (the_V2Label_of G).x = (the_V2Label_of LexBFS:LabelAdjacent(G,v,k)).x;

theorem :: LEXBFS:34
for G being VVGraph, v,x being set, k being natural number
 st x in G.AdjacentSet((v)) & not x in dom (the_VLabel_of G)
  holds (the_V2Label_of LexBFS:LabelAdjacent(G,v,k)).x =
        (the_V2Label_of G).x \/ {k};

theorem :: LEXBFS:35
for G being VVGraph, v being set, k being natural number
 st dom (the_V2Label_of G) = the_Vertices_of G
  holds dom (the_V2Label_of LexBFS:LabelAdjacent(G,v,k)) = the_Vertices_of G;

definition let G be finite natural-vlabeled finite-v2labeled
                    natsubset-v2labeled VVGraph,
             v be Vertex of G, k be natural number;
  redefine func LexBFS:LabelAdjacent(G, v, k) ->
                    finite natural-vlabeled
                    finite-v2labeled natsubset-v2labeled VVGraph;
end;

definition let G be finite natural-vlabeled
                    finite-v2labeled natsubset-v2labeled VVGraph,
             v be Vertex of G, n be natural number;
   func LexBFS:Update(G, v, n) ->
            finite natural-vlabeled
            finite-v2labeled natsubset-v2labeled VVGraph equals
:: LEXBFS:def 31
     LexBFS:LabelAdjacent(G.labelVertex(v, G.order()-'n), v, G.order()-'n);
end;

definition let G be finite natural-vlabeled
                    finite-v2labeled natsubset-v2labeled VVGraph;
  func LexBFS:Step(G) ->
          finite natural-vlabeled
          finite-v2labeled natsubset-v2labeled VVGraph equals
:: LEXBFS:def 32
   G if G.order() <= card (dom the_VLabel_of G)
             otherwise LexBFS:Update(G,
                                     LexBFS:PickUnnumbered(G),
                                     card (dom the_VLabel_of G));
end;

definition let G be finite _Graph;
  func LexBFS:CSeq(G) ->
       finite natural-vlabeled
       finite-v2labeled natsubset-v2labeled VVGraphSeq means
:: LEXBFS:def 33
 it.0 = LexBFS:Init(G) &
        for n being natural number holds it.(n+1) = LexBFS:Step(it.n);
end;

theorem :: LEXBFS:36
for G being finite _Graph holds LexBFS:CSeq(G) is iterative;

registration let G be finite _Graph;
  cluster LexBFS:CSeq(G) -> iterative;
end;

theorem :: LEXBFS:37  :: tLexBFSINIT01:
for G being _Graph holds the_VLabel_of LexBFS:Init(G) = {};

theorem :: LEXBFS:38  :: tLexBFSINIT02:
for G being _Graph, V being set
  holds dom the_V2Label_of LexBFS:Init(G) = the_Vertices_of G &
        (the_V2Label_of LexBFS:Init(G)).V = {};

theorem :: LEXBFS:39  :: tLexBFSINIT03:
for G being _Graph holds G <= LexBFS:Init(G);

:: the vertex picked has the largest v2label
theorem :: LEXBFS:40  :: tLexBFSPick01:
for G being finite finite-v2labeled natsubset-v2labeled VVGraph, x being set
 st not x in dom the_VLabel_of G &
    dom the_V2Label_of G = the_Vertices_of G &
    dom the_VLabel_of G <> the_Vertices_of G
 holds ((the_V2Label_of G).x,1) -bag <=
       ((the_V2Label_of G).(LexBFS:PickUnnumbered(G)),1) -bag, InvLexOrder NAT;

:: the vertex picked is not currently in the vlabel
theorem :: LEXBFS:41  :: tLexBFSPick02:
for G being finite finite-v2labeled natsubset-v2labeled VVGraph
 st dom the_V2Label_of G = the_Vertices_of G &
    dom the_VLabel_of G <> the_Vertices_of G
 holds not LexBFS:PickUnnumbered(G) in dom the_VLabel_of G;

theorem :: LEXBFS:42
for G being finite _Graph, n being natural number
  holds (LexBFS:CSeq(G)).n == G;

:: show lexbfs has static vertices
theorem :: LEXBFS:43
for G being finite _Graph, m,n being natural number
```

```
  holds (LexBFS:CSeq(G)).m <> (LexBFS:CSeq(G)).n;

theorem :: LEXBFS:44
for G being finite _Graph, n being natural number
 st card (dom the_VLabel_of ((LexBFS:CSeq(G)).n)) < G.order()
holds
   the_VLabel_of ((LexBFS:CSeq(G)).(n+1)) =
   (the_VLabel_of ((LexBFS:CSeq(G)).n)) +*
   (LexBFS:PickUnnumbered((LexBFS:CSeq(G)).n) .-->
      (G.order()-'(card (dom the_VLabel_of ((LexBFS:CSeq(G)).n)))));

theorem :: LEXBFS:45
for G being finite _Graph, n being natural number
 holds dom (the_V2Label_of ((LexBFS:CSeq(G)).n)) =
           the_Vertices_of ((LexBFS:CSeq(G)).n);

theorem :: LEXBFS:46
for G being finite _Graph, n being natural number st n <= G.order()
 holds card dom the_VLabel_of ((LexBFS:CSeq(G)).n) = n;

theorem :: LEXBFS:47
for G being finite _Graph, n being natural number st G.order() <= n
  holds (LexBFS:CSeq(G)).(G.order()) = (LexBFS:CSeq(G)).n;

theorem :: LEXBFS:48
for G being finite _Graph, m,n being natural number st G.order() <= m & m <= n
  holds (LexBFS:CSeq(G)).m = (LexBFS:CSeq(G)).n;

theorem :: LEXBFS:49
for G being finite _Graph holds LexBFS:CSeq(G) is eventually-constant;

registration let G be finite _Graph;
  cluster LexBFS:CSeq(G) -> eventually-constant;
end;

theorem :: LEXBFS:50
for G being finite _Graph, n being natural number holds
   dom the_VLabel_of((LexBFS:CSeq(G)).n) = the_Vertices_of((LexBFS:CSeq(G)).n)
iff G.order() <= n;

theorem :: LEXBFS:51
for G being finite _Graph holds (LexBFS:CSeq(G)).Lifespan() = G.order();

registration let G be finite chordal _Graph, i be natural number;
  cluster (LexBFS:CSeq(G)).i -> chordal VVGraph;
end;

registration
  let G be finite chordal _Graph;
  cluster LexBFS:CSeq(G) -> chordal;
end;

theorem :: LEXBFS:52
for G being finite _Graph holds LexBFS:CSeq(G) is vlabel-numbering;

registration let G be finite _Graph;
  cluster LexBFS:CSeq(G) -> vlabel-numbering;
end;
```

```
theorem :: LEXBFS:53
for G being finite _Graph, n being natural number st n < G.order()
  holds (LexBFS:CSeq(G)).PickedAt(n) =
        LexBFS:PickUnnumbered((LexBFS:CSeq(G)).n);

theorem :: LEXBFS:54
for G being finite _Graph, n being natural number st n < G.order()
 ex w being Vertex of (LexBFS:CSeq(G)).n
   st w = LexBFS:PickUnnumbered((LexBFS:CSeq(G)).n) &
   for v being set holds
    ((v in G.AdjacentSet((w)) &
      not v in dom (the_VLabel_of ((LexBFS:CSeq(G)).n))
      implies
       (the_V2Label_of ((LexBFS:CSeq(G)).(n+1))).v =
       (the_V2Label_of ((LexBFS:CSeq(G)).n)).v \/ {G.order() -' n}) &
    (not v in G.AdjacentSet((w)) or
         v in dom (the_VLabel_of ((LexBFS:CSeq(G)).n)))
       implies
        (the_V2Label_of ((LexBFS:CSeq(G)).(n+1))).v =
        (the_V2Label_of ((LexBFS:CSeq(G)).n)).v);

theorem :: LEXBFS:55
for G being finite _Graph, i being natural number, v being set holds
(the_V2Label_of ((LexBFS:CSeq(G)).i)).v c= (Seg G.order()) \ Seg (G.order()-'i)
;

:: LexBFS: Property L1
theorem :: LEXBFS:56
for G being finite _Graph, x being set, i,j being natural number st i <= j
 holds     (the_V2Label_of ((LexBFS:CSeq(G)).i)).x
       c= (the_V2Label_of ((LexBFS:CSeq(G)).j)).x;

theorem :: LEXBFS:57
for G being finite _Graph, m,n being natural number, x, y being set
 st n < G.order() & m < n & y = LexBFS:PickUnnumbered((LexBFS:CSeq(G)).n) &
    not x in dom (the_VLabel_of ((LexBFS:CSeq(G)).m)) & x in G.AdjacentSet((y))
  holds (G.order() -' m) in (the_V2Label_of ((LexBFS:CSeq(G)).m)).x;

theorem :: LEXBFS:58
for G being finite _Graph, m,n being natural number st m < n
for x being set
 st not G.order()-'m in (the_V2Label_of ((LexBFS:CSeq(G)).(m+1))).x
  holds not G.order() -' m in (the_V2Label_of ((LexBFS:CSeq(G)).n)).x;

:: More general version of the above:
:: if the value added during step k doesn't appear in a later step (n),
:: then that value cannot appear in an even later step (m)
theorem :: LEXBFS:59
for G being finite _Graph, m,n,k being natural number st k < n & n <= m
for x being set
 st not G.order()-'k in (the_V2Label_of ((LexBFS:CSeq(G)).n)).x
  holds not G.order()-'k in (the_V2Label_of ((LexBFS:CSeq(G)).m)).x;

:: relates a value in a vertex's v2label to the vertex chosen at that time
theorem :: LEXBFS:60
for G being finite _Graph, m,n being natural number
for x being Vertex of ((LexBFS:CSeq(G)).m)
  st n in (the_V2Label_of ((LexBFS:CSeq(G)).m)).x
    ex y being Vertex of ((LexBFS:CSeq(G)).m)
```

```
    st ((LexBFS:PickUnnumbered((LexBFS:CSeq(G)).(G.order()-'n'))) = y &
      not y in dom (the_VLabel_of ((LexBFS:CSeq(G)).(G.order()-'n'))) &
      x in G.AdjacentSet({y}));

definition let Gs be finite natural-vlabeled VVGraphSeq;
  redefine func Gs.Result() -> finite natural-vlabeled VVGraph;
end;

theorem :: LEXBFS:61
for G being finite _Graph holds
    ((LexBFS:CSeq(G)).Result()).labeledV() = the_Vertices_of G;

theorem :: LEXBFS:62  :: tLexBFS05:
for G being finite _Graph
 holds (the_VLabel_of (LexBFS:CSeq(G)).Result())" is VertexScheme of G;

:: A vertex with a vlabel of k must have had the largest vlLabel when chosen
theorem :: LEXBFS:63
for G being finite _Graph, i,j being natural number,
    a,b being Vertex of (LexBFS:CSeq(G)).i
 st a in dom the_VLabel_of ((LexBFS:CSeq(G)).i) &
    b in dom the_VLabel_of ((LexBFS:CSeq(G)).i) &
    (the_VLabel_of ((LexBFS:CSeq(G)).i)).a <
                                    (the_VLabel_of ((LexBFS:CSeq(G)).i)).b &
    j = G.order() -' (the_VLabel_of ((LexBFS:CSeq(G)).i)).b
  holds ((the_V2Label_of ((LexBFS:CSeq(G)).j)).a,1)-bag <=
        ((the_V2Label_of ((LexBFS:CSeq(G)).j)).b,1)-bag, InvLexOrder NAT;

:: Any value in our v2label corresponds to a vertex that we are
:: adjacent to in our in our vlabel
theorem :: LEXBFS:64
for G being finite _Graph, i,j being natural number,
    v being Vertex of (LexBFS:CSeq(G)).i
 st j in (the_V2Label_of ((LexBFS:CSeq(G)).i)).v
  ex w being Vertex of (LexBFS:CSeq(G)).i st
     w in dom the_VLabel_of ((LexBFS:CSeq(G)).i) &
     (the_VLabel_of ((LexBFS:CSeq(G)).i)).w = j & v in G.AdjacentSet(w);

definition let G be natural-vlabeled VGraph;
  attr G is with_property_L3 means
:: LEXBFS:def 34
for a,b,c being Vertex of G st a in dom the_VLabel_of G &
    b in dom the_VLabel_of G & c in dom the_VLabel_of G &
    (the_VLabel_of G).a < (the_VLabel_of G).b &
    (the_VLabel_of G).b < (the_VLabel_of G).c &
    a,c are_adjacent & not b,c are_adjacent
  ex d being Vertex of G st d in dom the_VLabel_of G &
     (the_VLabel_of G).c < (the_VLabel_of G).d &
     b,d are_adjacent & not a,d are_adjacent &
     for e being Vertex of G
       st e <> d & e,b are_adjacent & not e,a are_adjacent
        holds (the_VLabel_of G).e < (the_VLabel_of G).d;
end;

theorem :: LEXBFS:65
for G being finite _Graph, n being natural number
  holds (LexBFS:CSeq(G)).n is with_property_L3;

theorem :: LEXBFS:66  :: Theorem 4.3, Golumbic p. 88
for G being finite chordal natural vlabeled VVGraph
  st G is with_property_L3 & dom the_VLabel_of G = the_Vertices_of G
   for V being VertexScheme of G st V" = the_VLabel_of G holds V is perfect;

theorem :: LEXBFS:67 :: Theorem 4.3, Golumbic p. 86
for G being finite chordal VVGraph holds
  (the_VLabel_of (LexBFS:CSeq(G)).Result())" is perfect VertexScheme of G;

begin :: The Maximum Cardinality Search algorithm

definition let G be finite _Graph;
  func MCS:Init(G) -> finite natural vlabeled natural v2labeled VVGraph equals
:: LEXBFS:def 35
   G.set(VLabelSelector, {}).set(V2LabelSelector, the_Vertices_of G  --> 0);
end;

definition let G be finite natural-v2labeled VVGraph;
  assume  dom the_V2Label_of G = the_Vertices_of G;
  func MCS:PickUnnumbered(G) -> Vertex of G means
:: LEXBFS:def 36
  it = choose the_Vertices_of G if dom the_VLabel_of G = the_Vertices_of G
  otherwise ex S being finite non empty natural-membered set, F being Function
              st S = rng F
           & F = (the_V2Label_of G) | (the_Vertices_of G \ dom the_VLabel_of G)
           & it = choose (F " (max S));
end;

definition let G be finite natural-v2labeled VVGraph, v be set;
 func MCS:LabelAdjacent(G, v) -> finite natural-v2labeled VVGraph equals
:: LEXBFS:def 37
 G.set(V2LabelSelector, (the_V2Label_of G).incSubset(G.AdjacentSet({v}))
                                                  \ dom (the_VLabel_of G),1));
end;

definition let G be finite natural-vlabeled natural-v2labeled VVGraph,
              v be Vertex of G;
  redefine func MCS:LabelAdjacent(G, v) -> finite natural-vlabeled
                                           natural-v2labeled VVGraph;
end;

definition
  let G be finite natural-vlabeled natural-v2labeled VVGraph,
      v be Vertex of G, n be natural number;
  func MCS:Update(G, v, n) -> finite natural-vlabeled natural-v2labeled
                              VVGraph equals
:: LEXBFS:def 38
      MCS:LabelAdjacent(G.labelVertex(v, G.order()-'n), v);
end;

definition let G be finite natural-vlabeled natural v2labeled VVGraph;
  func MCS:Step(G)
       -> finite natural-vlabeled natural-v2labeled VVGraph equals
:: LEXBFS:def 39
          G if G.order() <= card (dom the_VLabel_of G)
            otherwise MCS:Update(G,
                                 MCS:PickUnnumbered(G),
                                 card (dom the_VLabel_of G));
end;
```

```
definition let G be finite _Graph;
  func MCS:CSeq(G)
      -> finite natural-vlabeled natural-v2labeled VVGraphSeq means
:: LEXBFS:def 40
       it.0 = MCS:Init(G) &
            for n being natural number holds it.(n+1) = MCS:Step(it.n);
end;

theorem :: LEXBFS:68
for G being finite _Graph holds MCS:CSeq(G) is iterative;

registration let G be finite _Graph;
  cluster MCS:CSeq(G) -> iterative;
end;

theorem :: LEXBFS:69
for G being finite _Graph holds the_VLabel_of MCS:Init(G) = {};

theorem :: LEXBFS:70
for G being finite _Graph, v being set
  holds dom the_V2Label_of MCS:Init(G) = the_Vertices_of G &
        (the_V2Label_of MCS:Init(G)).v = 0;

theorem :: LEXBFS:71
for G being finite _Graph holds G == MCS:Init(G);

theorem :: LEXBFS:72
for G being finite natural-v2labeled VVGraph, x being set
 st not x in dom the_VLabel_of G &
    dom the_V2Label_of G = the_Vertices_of G &
    dom the_VLabel_of G <> the_Vertices_of G
 holds (the_V2Label_of G).x <= (the_V2Label_of G).(MCS:PickUnnumbered(G));

theorem :: LEXBFS:73
for G being finite natural-v2labeled VVGraph
 st dom the_V2Label_of G = the_Vertices_of G &
    dom the_VLabel_of G <> the_Vertices_of G
 holds not MCS:PickUnnumbered(G) in dom the_VLabel_of G;

theorem :: LEXBFS:74
for G being finite natural-v2labeled VVGraph, v,x being set
 st not x in G.AdjacentSet({v})
  holds (the_V2Label_of G).x = (the_V2Label_of (MCS:LabelAdjacent(G,v))).x;

theorem :: LEXBFS:75
for G being finite natural-v2labeled VVGraph, v,x being set
 st x in dom (the_VLabel_of G)
  holds (the_V2Label_of G).x = (the_V2Label_of (MCS:LabelAdjacent(G,v))).x;

theorem :: LEXBFS:76
for G being finite natural-v2labeled VVGraph, v,x being set
 st x in dom the_V2Label_of G &
    x in G.AdjacentSet(v) & not x in dom the_VLabel_of G
  holds (the_V2Label_of (MCS:LabelAdjacent(G,v))).x = (the_V2Label_of G).x + 1;

theorem :: LEXBFS:77
for G being finite natural-v2labeled VVGraph, v being set
 st dom (the_V2Label_of G) = the_Vertices_of G
  holds dom (the_V2Label_of (MCS:LabelAdjacent(G,v))) = the_Vertices_of G;

theorem :: LEXBFS:78
for G being finite _Graph, n being natural number holds (MCS:CSeq(G)).n == G;

theorem :: LEXBFS:79
for G being finite _Graph, m, n being natural number
 holds (MCS:CSeq(G)).m == (MCS:CSeq(G)).n;

registration let G be finite chordal _Graph, n be natural number;
  cluster (MCS:CSeq(G)).n -> chordal VVGraph;
end;

registration let G be finite chordal _Graph;
  cluster MCS:CSeq(G) -> chordal;
end;

theorem :: LEXBFS:80
for G being finite _Graph, n being natural number holds
  dom (the_V2Label_of ((MCS:CSeq(G)).n)) = the_Vertices_of ((MCS:CSeq(G)).n);

theorem :: LEXBFS:81
for G being finite _Graph, n being natural number
  st card (dom the_VLabel_of ((MCS:CSeq(G)).n)) < G.order()
 holds the_VLabel_of ((MCS:CSeq(G)).(n+1)) =
       (the_VLabel_of ((MCS:CSeq(G)).n)) +*
                  (MCS:PickUnnumbered((MCS:CSeq(G)).n) .->
                  (G.order()-'card (dom the_VLabel_of ((MCS:CSeq(G)).n)))));

theorem :: LEXBFS:82
for G being finite _Graph, n being natural number st n <= G.order()
 holds card dom the_VLabel_of ((MCS:CSeq(G)).n) = n;

theorem :: LEXBFS:83
for G being finite _Graph, n being natural number st G.order() <= n
  holds (MCS:CSeq(G)).(G.order()) = (MCS:CSeq(G)).n;

theorem :: LEXBFS:84
for G being finite _Graph, m,n being natural number st G.order() <= m & m <= n
  holds (MCS:CSeq(G)).m = (MCS:CSeq(G)).n;

theorem :: LEXBFS:85
for G being finite _Graph holds MCS:CSeq(G) is eventually-constant;

registration let G be finite _Graph;
  cluster MCS:CSeq(G) -> eventually-constant;
end;

theorem :: LEXBFS:86
for G being finite _Graph, n being natural number holds
       dom the_VLabel_of ((MCS:CSeq(G)).n) = the_Vertices_of ((MCS:CSeq(G)).n)
   iff G.order() <= n;

theorem :: LEXBFS:87
for G being finite _Graph holds (MCS:CSeq(G)).Lifespan() = G.order();

theorem :: LEXBFS:88
for G being finite _Graph holds MCS:CSeq(G) is vlabel-numbering;

registration let G be finite _Graph;
```

```
  cluster MCS:CSeq(G) -> vlabel-numbering;
end;

theorem :: LEXBFS:89
for G being finite _Graph, n being natural number st n < G.order()
  holds (MCS:CSeq(G)).PickedAt(n) = MCS:PickUnnumbered((MCS:CSeq(G)).n);

theorem :: LEXBFS:90
for G being finite _Graph, n being natural number st n < G.order()
 ex w being Vertex of (MCS:CSeq(G)).n
   st w = MCS:PickUnnumbered((MCS:CSeq(G)).n) &
   for v being set holds
    (v in G.AdjacentSet({w}) & not v in dom (the_VLabel_of ((MCS:CSeq(G)).n))
       implies (the_V2Label_of ((MCS:CSeq(G)).(n+1))).v =
                                        (the_V2Label_of ((MCS:CSeq(G)).n)).v + 1)
  & (not v in G.AdjacentSet({w}) or v in dom (the_VLabel_of ((MCS:CSeq(G)).n))
       implies (the_V2Label_of ((MCS:CSeq(G)).(n+1))).v =
                                        (the_V2Label_of ((MCS:CSeq(G)).n)).v);

theorem :: LEXBFS:91
for G being finite _Graph, n being natural number, x being set
 st not x in (dom the_VLabel_of ((MCS:CSeq(G)).n))
  holds (the_V2Label_of ((MCS:CSeq(G)).n)).x =
        card (G.AdjacentSet({x}) /\ (dom the_VLabel_of ((MCS:CSeq(G)).n)));

definition let G be natural-vlabeled VGraph;
  attr G is with_property_7 means
:: LEXBFS:def 41
for a,b,c being Vertex of G st a in dom the_VLabel_of G &
    b in dom the_VLabel_of G & c in dom the_VLabel_of G &
    (the_VLabel_of G).a < (the_VLabel_of G).b &
    (the_VLabel_of G).b < (the_VLabel_of G).c &
    a,c are_adjacent & not b,c are_adjacent
  ex d being Vertex of G st d in dom the_VLabel_of G &
     (the_VLabel_of G).b < (the_VLabel_of G).d &
     b,d are_adjacent & not a,d are_adjacent;
end;

theorem :: LEXBFS:92
for G being finite _Graph, n being natural number
  holds (MCS:CSeq(G)).n is with_property_7;

theorem :: LEXBFS:93  :: LexBFS also has property 7
for G being finite _Graph
  holds (LexBFS:CSeq(G)).Result() is with_property_7;

theorem :: LEXBFS:94 :: Terjan (SIAM Journal of Computing; 13(3):August 1984)
for G being finite chordal natural-vlabeled VGraph
  st G is with_property_7 & dom the_VLabel_of G = the_Vertices_of G
   for V being VertexScheme of G st V" = the_VLabel_of G holds V is perfect;
```